UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

IMPLEMENTACIJA UČINKOVITEGA SISTEMA ZA GRADNJO, UPORABO IN EVALVACIJO LEMATIZATORJEV TIPA RDR

MATJAŽ JURŠIČ

Delo je pripravljeno v skladu s Pravilnikom o podeljevanju
Prešernovih nagrad študentom, pod mentorstvom
prof. dr. Blaža Zupana in somentorstvom prof. dr. Nade Lavrač.

LJUBLJANA, 2007

POVZETEK

V pričujočem delu smo zasnovali in implementirali zbirko orodji s področja lematizacije besedil v poljubnih jezikih. Izhajamo iz dveh naborov podatkov: iz množice primerov že lematiziranih besed ter iz besedil, ki jih želimo lematizirati. Naš končni cilj je lematizacija teh besedil. Celoten sistem smo razdelili na tri sklope, ki predstavljajo zaključene modularne enote. V prvem koraku se na množici primerov lematiziranih besed naučimo pravil, ki kar najbolje opisujejo njihovo lematizacijo. Ta pravila so predstavljena v obliki RDR dreves. Naslednji sklop iz teh dreves izdela izredno učinkovito strukturo za lematizacijo oz. lematizator. V zadnjem koraku ta lematizator uporabimo za lematizacijo izhodiščnih besedil. Sistem smo testirali glede točnosti učenja in hitrosti lematizacije v primerjavi z obstoječim sistemom. Testiranje na velikih večjezičnih leksikonih Multext in Multext-East je pokazalo znatno izboljšanje hitrosti in točnosti lematizatorja. Naredili smo tudi aplikacijo na agencijskih novicah, ki služijo kot vhod metodi odkrivanja znanj iz besedil. Na besedilih novic smo pokazali, da se uporabnost te metode s predhodno aplikacijo lematizacije zelo poveča.

KLJUČNE BESEDE

- lematizacija
- RDR (ripple down rules)
- predobdelava besedil
- analiza besedil
- odkrivanje znanj v podatkih

ABSTRACT

Lemmatization is the process of determining the canonical form of a word, called lemma, from its inflectional variants. We have developed a language independent system, LemmaGen, consisting of a set of tools for automatically learning of lemmatizers from lexicons of pre-lemmatized words. The system consists of three modules that can be used independently or sequentially. The input to the first module is a lexicon of lemmatized words from which it learns Ripple Down Rules that best describe word lemmatization. The next module takes these rules, which are in the form of RDR trees, and produces an efficient structure for fast lemmatization - the actual lemmatizer. In the last step we use the lemmatizer to transform the original input text into a set of lemmatized words. LemmaGen was applied to 14 different Multext and Multext-East lexicons and produced efficient lemmatizers for the corresponding languages. Its evaluation on the 14 lexicons shows that LemmaGen considerably outperforms the lemmatizers generated by the previously developed RDR learning algorithm, both in terms of accuracy and efficiency. We used lemmatization also as a step in the analysis of a corpus of press-agency news and show improved result interpretation, achieved by using LemmaGen in news preprocessing.

KEYWORDS

- Lemmatization
- RDR (Ripple Down Rules)
- Text preprocessing
- Text mining
- Knowledge discovery

KAZALO

1 UVOD

Lematizacija je postopek pretvarjanja besede v njeno lemo, tj. nevtralno oz. slovarsko obliko te besede. Uporablja se predvsem na področju klasifikacije ter iskanja besedil. Ta tematika nas je pritegnila, ker za slovenščino nismo imeli prosto dostopnega programa, ki bi nam nudil kakovostno lematizacijo in bi bil dovolj zanesljiv za obdelavo naših podatkov. Kratica RDR, ki se pojavlja v naslovu *(angl. Ripple Down Rules)*, opisuje metodologijo, po kateri se naučimo pravil za lematizacijo. Čeprav smo se primarno ukvarjali s slovenščino, je možno vsa dognanja iz tega dela, uporabljati tudi na drugih jezikih.

1.1 LEMATIZACIJA

Lematizacija *(angl. lemmatization)* je postopek, s katerim besedam določimo njihove leme *(angl. lemma)*. Lema besede ali natančnejše besedne enote je njena osnovna morfološka različica. Leme si lahko predstavljamo kot oblike besed, ki jih najdemo v slovarju. V različnih jezikih so leme za različne besedne vrste različno določene. V slovenščini imajo npr. glagoli za lemo nedoločnik, samostalniki pa sklanjatveno obliko v imenovalniku ednine. Primer različnih morfoloških oblik besedne enote *pisati so: pisati, pišem, pišeš, piše, piševa, pišeta, pišeta,...;* celoten seznam je naveden v primeru 5.1. V SSKJ je lematizacija opredeljena kot: »določanje besednih enot za gesla ali podgesla v slovarju, enciklopediji, gesljenje«. Zgled lematizacije stavka si lahko ogledate v primeru 1.1.

Pri tvorbi stavkov besede večinoma dobijo neko novo, spregano, sklanjano ali drugače spremenjeno morfološko obliko. Ta postopek imenujemo pregibanje besed. Jeziki se glede stopnje pregibnosti močno razlikujejo. Slovenščina je primer bogato pregibnega jezika, saj imajo besede lahko tudi do 30 različnih pregibnih oblik. Primer jezika z nizko stopnjo pregibnosti je angleščina, kjer imajo besede po večini le nekaj oblik. Na lematizacijo lahko gledamo tudi kot na obratni postopek od pregibanja. Zaradi tega je lematizacija za bolj pregibne jezike pomembnejša in praviloma tudi kompleksnejša. V skupini indo-evropskih jezikov, kamor spadajo tudi večje podskupine kot na primer germanski, slovanski in romanski jeziki, se pregibanje večinoma izraža kot spreminjanje končnice

besed. Postopki, s katerimi se ukvarjamo v tem delu, so primerni le za jezike s takšnim tipom pregibanja.

Lematizaciji podoben postopek se imenuje krnjenje *(angl. stemming).* Pri krnjenju različnim besednim oblikam ne določamo leme ampak koren oz. krn *(angl. stem),* tj. nespremenljiv del vseh njenih morfoloških oblik. Lematizacija je zahtevnejša kot krnjenje, vendar je za večino aplikacij bolj primerna. Pri slednjem se namreč lahko več besed zlije v isti koren in tako pride do izgube informacije v besedilu. Primer korena besede *pisati* je *pi*. V tej diplomi se s krnjenjem sicer ne ukvarjamo, vendar je možno vse postopke brez prilagoditev uporabiti tudi v ta namen. Navsezadnje je krnjenje le okrnjena lematizacija.

PRIMER 1.1: LEMATIZACIJA STAVKA

1	Nesel je Krpan po ozki gazi na svoji kobilici nekoliko stotov soli; kar mu naproti prižvenketa lep voz; na vozu je pa sedel cesar Janez, ki se je ravno peljal v Trst.
2	Nesti biti Krpan po ozek gaz na svoj kobilica nekolik stot sol; kar on naproti prižvenketati lep voz; na voz biti pa sedeti cesar Janez, ki se biti ravno peljati v Trst.

Na tem mestu velja omeniti še, da včasih lematizacijo definirajo tudi kot postopek za pretvorbo besed v korene z uporabo informacije o vlogi besede v stavku *(angl. part of speech)*. V tem delu to ne velja, saj te informacije ne uporabljamo. Naša definicija je enaka kot v [11]; lematizacija je torej zamenjava morfološke končnice s končnico leme te besedne enote.

Program oz. algoritem, katerega naloga je lematizacija besed določenega jezika, imenujemo lematizator *(angl. lemmatizer)*.

Tipična primera uporabe lematizacije sta spletni iskalnik ter odkrivanje znanj v besedilih. V teh primerih gre za klasičen problem iskanja ključnih besed neke spletne strani oz. dokumenta. Najpreprostejši način iskanja ključnih besed je izbira statistično najpogostejših besed. Vendar, če se dejanske ključne besede nahajajo v veliko morfoloških oblikah, lahko zgrešimo tiste prave, saj ne bodo dovolj pogoste. V takih primerih moramo vedno najprej narediti lematizacijo besedil ter v primeru iskalnikov tudi iskanih pojmov.

1.2 PREGLED METOD LEMATIZACIJE

Problem lematizacije oz. krnjenja je za računalniško znanost razmeroma star. Prvi članek s tega področja [1] je bil objavljen že leta 1968. Za določene jezike se ta domena smatra kot zaključena in se z njo večinoma ne ukvarjajo več. Za angleščino je problem z zadostno točnostjo rešil M.F. Porter, ki je leta 1980 objavil algoritem za krnjenje [14], ki je kasneje postal de-facto standard za lematizacijo angleških besedil. Za večino ostalih jezikov, predvsem takih z večjo pregibnostjo, pa se raziskave še niso zaključile. Med te jezike spada tudi slovenščina.

Lematizator lahko izdelamo na dva načina, ročno ali avtomatsko. Prvega naredimo tako, da jezikoslovec ali ekspert s tega področja določi pravila, po katerih poteka lematizacija. Ta pravila nato zakodiramo v lematizator. Druga, iz perspektive slovenščine privlačnejša metoda, pa je avtomatska gradnja lematizatorja. V ta namen uporabimo nek postopek strojnega učenja [8], katerega na množici že lematiziranih besed naučimo lematizacije. Rezultat takega učenja pogosto le z malo truda spremenimo v algoritem lematizatorja. Tudi za avtomatsko zgrajene programe je zaželeno, da so transparentni. To pomeni, da so uporabniku na voljo naučena pravila, ki lahko tudi pojasnijo, zakaj so določeno besedo lematizirala na določen način. Tako lahko eksperti po potrebi preverijo in analizirajo delovanje lematizatorja.

Porterjev algoritem spada v kategorijo ročno izdelanih lematizatorjev. Ker pa je ročna izdelava lematizatorjev za pregibnejše jezike preveč zahtevna, poteka v zadnjem času razvoj v okviru avtomatsko generiranih lematizatorjev.

Številni strokovni članki s tega področja pričajo, da je problematika še vedno aktualna. Učenje pravil za lematizacijo naravno spada med metode umetne inteligence, zato vse rešitve, ki smo jih našli, izhajajo s tega področja. Mnogi raziskovalci so se reševanja lotili na različne načine. Tu navajamo le nekaj glavnih pristopov:

- 1993-2002 Sistem za indukcijo pravil ATRIS [11; 10]
- 2002 If-then klasifikacijska pravila [9]
- 2002 Naivni Bayes [9]
- 2004 Sistem za učenje odločitvenih pravil prvega reda CLog [4]
- 2004 Sistem za učenje pravil RDR [12; 13]

1.3 METODOLOGIJA RDR

RDR je metodologija inkrementalne ekstrakcije iz podatkov. RDR sta predlagala Compton in Jansen upoštevajoč njune izkušnje z vzdrževanjem ekspertnega sistema GARVAN-ES1 [3]. Koncept RDR poskuša posnemati učenje strokovnjakov določene domene, kjer se znanje dodaja inkrementalno. Ker ekspert izdela pravilo na podlagi lastnosti trenutnega primera, njegovega razreda in poznavanja domene, je malo verjetno, da bo pravilo pravilno klasificiralo vse primere tega razreda.

Sistem se je naučil znanja na začetni množici primerov, ko pa so bili na voljo novi, je sistem preveril njihovo klasifikacijo. Če je bila napačna, potem so bili primeri uporabljeni za dodajanje novega znanja in njegovo inkrementalno izboljšanje. Bazo pravil je bilo potrebno po vsakem dodajanju pravil znova preveriti, saj so se lahko pojavili konflikti oz. nekonsistence med pravili.

Čeprav so RDR *(Ripple Down Rules)* pravila, pa je podatkovna struktura RDR drevo pravil ter njihovih izjem. Enostavna struktura je prikazana v primeru 1.2 (povzeto iz [12]). Odločanje v tem

drevesu poteka na naslednji način. Če primer ustreza pogoju A in B potem algoritem vrne sklep C, razen če primer izpolnjuje tudi pogoj D, takrat vrne sklep E. Če pogoj A in B ni izpolnjen, potem algoritem preveri naslednji pogoj F in D. Če je ta pogoj izpolnjen, potem vrne sklep H.

PRIMER 1.2: ENOSTAVNO RDR DREVO

```
1  if (A and B) then C
2      except if D then E
3  else if (F and D) then H
```

KODA 1.1: PSEVDOKODA ISKANJA PRAVILA, KI GA PROŽI DANI PRIMER

```
1  FindFiredRule(currRule, example)
2      foreach (excRule in currRule.exceptionList)
3          if (excRule.Condition(example) == true)
4              return FindFiredRule(excRule, example)
5      return currRule
```

V splošnem je koren drevesa pravilo brez pogojev in ga prožijo *(angl. fire)* vsi primeri. Če pravilo ni končno, tj. list v drevesu, potem so njemu podrejena pravila definirana kot njegove izjeme. Čim globlje se spuščamo po drevesu, bolj specifični so pogoji pravil. Ko iščemo pravilo, ki velja oz. ga proži dani primer, nas vedno zanima le najbolj specifično pravilo. Ko želimo klasificirati nov primer, je potrebno le najti pravilo, ki ga le ta proži. Algoritem iskanja pravila opisuje koda 1.1. Pri učenju je postopek pravzaprav zelo podoben. Najprej najdemo pravilo, ki ga proži trenutni učni primer. Če je klasifikacija tega pravila ustrezna, potem ne naredimo nič, v nasprotnem primeru pa dodamo izjemo z dodatnimi lastnostmi tega učnega primera.

1.4 RDR V DOMENI LEMATIZACIJE

Če želimo postaviti metodologijo RDR v domeno lematizacije, moramo definirati določene neopredeljene lastnosti RDR mehanizma. Definicije, ki smo povzeli neposredno iz članka [12], so naslednje:

- V domeni lematizacije so primeri kar morfološke oblike besed.
- Razred primera oz. transformacija je najkrajši postopek, ki nam besedo iz morfološke oblike pretvori v njeno lemo. Najkrajši je mišljen tak, ki zamenja najmanjše število zadnjih črk besede. Transformacijo lahko zapišemo kot *{končnica morf.}->{končnica leme}*. Zgledi primerov z lemami ter pripadajočimi razredi so podani v primeru 1.3.
- Pogoj pravila je končnica *(angl. suffix),* s katero se mora končati beseda, da proži to pravilo. Iskalna funkcija iz kode 1.1 tako le primerja končnice besed s pogoji pravil.

PRIMER 1.3: ZGLEDI MORFOLOŠKIH OBLIK, NJIHOVIH LEM TER PRIPADAJOČIH TRANSFORMACIJ

	morf.oblika	lema	transformacija oz razred
1	pisala	pisati	'la'->'ti'
2	pisal	pisalo	''->'o'
3	pišete	pisati	'šete'->'sati'
4	pisalo	pisalo	''->''

V tem trenutku še ne moremo natančno opisati postopka učenja drevesa, podajamo pa primer RDR drevesa v domeni lematizacije (primer 1.4). Drevo je bilo generirano z algoritmom [12], ki je sicer bolj podrobno opredeljen v poglavju (4.2 Osnovni algoritem). Primer je reprezentativen in se nanj v nadaljnjih poglavjih pogosto sklicujemo, zato omenimo še, da je bil naučen na 10 označenih besedah primera 5.1. Te besede niso bile izbrane naključno, ampak smo jih skrbno določili tako, da lahko kljub izredno majhnemu drevesu pokažemo čim več njegovih lastnosti. V mislih smo imeli tudi navezujoče primere naslednjih poglavij.

Notacija dreves je preprosta. Ključna beseda *[RULE:]* začne vsako pravilo. Lastnost *[suffix]* definira končnico, ki je pogoj, da se pravilo proži, *[transform]* pa transformacijo, ki se izvede na besedi, če se pravilo proži. Podrejena vozlišča so dejansko izjeme nadrejenega. Natančneje je notacija razložena v poglavju (3.2 Določitev vhodne datoteke).

PRIMER 1.4: RDR DREVO GENERIRANO Z ALGORITMOM [12] IZ OZNAČENIH BESED PRIMERA 5.1

```
1      RULE:( suffix("") transform(""-->"") except(5) ); {:
1.1     |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
1.2     |---> RULE:( suffix("ši") transform("ši"-->"sati") );
1.3     |---> RULE:( suffix("šimo") transform("šimo"-->"sati") );
1.4     |---> RULE:( suffix("l") transform("l"-->"ti") );
1.5     `---> RULE:( suffix("i") transform("i"-->"o") except(2) ); {:
1.5.1          |---> RULE:( suffix("ni") transform("ni"-->"ti") );
1.5.2          `---> RULE:( suffix("ti") transform(""-->"") ); :}
                :}
```

1.5 MOTIVACIJA IN PRISPEVEK DELA

Z lematizacijo smo se začeli ukvarjati med obdelavo agencijskih novic o dogodkih v Sloveniji. Članki so bili povzeti od tiskovnih agencij različnih držav in zanimalo nas je, kako se glede na države ti članki razlikujejo. V vsaki državi dajo prednost drugim novicam iz Slovenije. Če posplošimo, bi lahko rekli, da smo skušali iz novic izluščiti podobo Slovenije, ki jo imajo prebivalci tistih držav, v katerih mediji poročajo o nas.

Orodje, ki nam je omogočilo pregled nad veliko količino člankov, se imenuje Ontogen [6]. Njegova naloga je odkrivanje znanj v besedilih in na podlagi tega gradnja ontologij. Ontlogija je neke vrste hierarhična ureditev vsebin oz tematik, o katerih govorijo obravnavana besedila. Kot pri vseh metodah odkrivanja znanj iz besedil je tudi pri Ontogenu poglavitnega pomena pravilno zaznavanje ključnih besed. V želji dobiti čim boljše rezultate je bilo zato potrebno narediti predhodno lematizacijo besedil, saj Ontogen še ni vseboval podatkov za lematizacijo slovenščine.

Zaradi razmeroma dobre točnosti lematizacije in preproste ideje smo se odločili, da implementiramo članek Plisson, J., in drugi: Pristop k lematizaciji besed z uporabo pravil [12]. Pravzaprav gre v diplomskem delu za nadgradnjo članka z določenimi praktičnimi rešitvami in učinkovito implementacijo nekaterih predlaganih zamisli. Ker smo poleg tega dobili še velike

leksikone lematiziranih besed [5], se je naša osnovna ideja počasi preoblikovala v tudi v učenje pravil za lematizacijo. Tudi tukaj smo se zgledovali po članku [12], omejili pa se nismo samo na slovenščino, saj smo imeli tudi podatke za 11 drugih evropskih jezikov. Rešili pa smo tudi osnovni problem lematizacije agencijskih člankov in gradnje ontologij. Več o tem si lahko preberete v poglavju "5.2 Aplikacija na agencijskih člankih".

Tako pridemo tudi do prispevkov tega dela. Izdelali smo sistem sestavljen iz treh sklopov, ki se je sposoben naučiti lematizacije, prikazati naučena pravila ter učinkovito lematizirati dana besedila. Zaradi modularne zgradbe je možna tudi lematizacija s pravili, ki niso plod našega učnega algoritma, ampak smo jih dobili drugje. Rešitev smo izvedli celostno, kar pomeni, da smo pokrili vse nivoje razvoja programske opreme od pregleda obstoječih rešitev prek zasnove ter implementacije do testiranja in aplikacije sistema. Na nekaj mestih smo dodali temu produkcijskemu pristopu tudi svoj lastni znanstveni prispevek. Tako smo na primer napisali popolnoma nov učni algoritem ter izdelali metodo za optimizacijo RDR dreves.

Rezultati naloge se lahko uporabljajo samostojno, za lematizacijo besedil, ali kot knjižnica metod, za vključitev v nek obširnejši sistem. Razviti moduli, pripadajoča izvorna koda ter ostale elektronske priloge so prosto dostopne, njihov opis pa podaja poglavje "Priloge B". V kolikor bo naš prispevek vključen v sistem Ontogen, to posledično pomeni bolj preprosto oz. kvalitetnejšo gradnjo ontologij iz slovenskih in tudi morda tudi določenih tujih besedil.

1.6 Struktura diplomskega dela

Diploma strukturno sledi sklopom oz. modulom sistema, ki smo ga izdelali. Uvodu sledijo tri poglavja, ki se neposredno navezujejo na tri glavne sklope. To delitev si lahko nazorno ogledamo v diagramu 1.1.

Poglavje "RDR lematizator" opiše končni cilj celotnega postopka gradnje lematizatorja. Tukaj postavimo zahteve in ogrodje razvitega lematizatorja. Definirana sta struktura podatkov in algoritem za lematizacijo, poudarek pa je na učinkovitem delovanju. V sklopu "Gradnja lematizatorjev" se ukvarjamo s problemom, kako iz poljubnega drevesa RDR pravil izdelati lematizator zasnovan v prejšnjem poglavju. Poglavje "Učenje RDR pravil" razloži implementacijo gradnje RDR pravil predlagane v [12] vendar na malo drugačen, izboljšan način. Ta tri poglavja tako zaokrožajo celoten krog gradnje lematizatorja iz učnih primerov ter njegove uporabe na novih besedilih.

V poglavju z rezultati podamo primerjavo algoritmov učenja in lematizacije ter pokažemo na razlike med različnimi jeziki. V drugem delu tega poglavja predstavimo tudi aplikacijo naučenega lematizatorja na našem začetnem problemu analize agencijskih novic. Diplomo zaključimo s povzetkom in pregledom možnosti za nadaljnje delo.

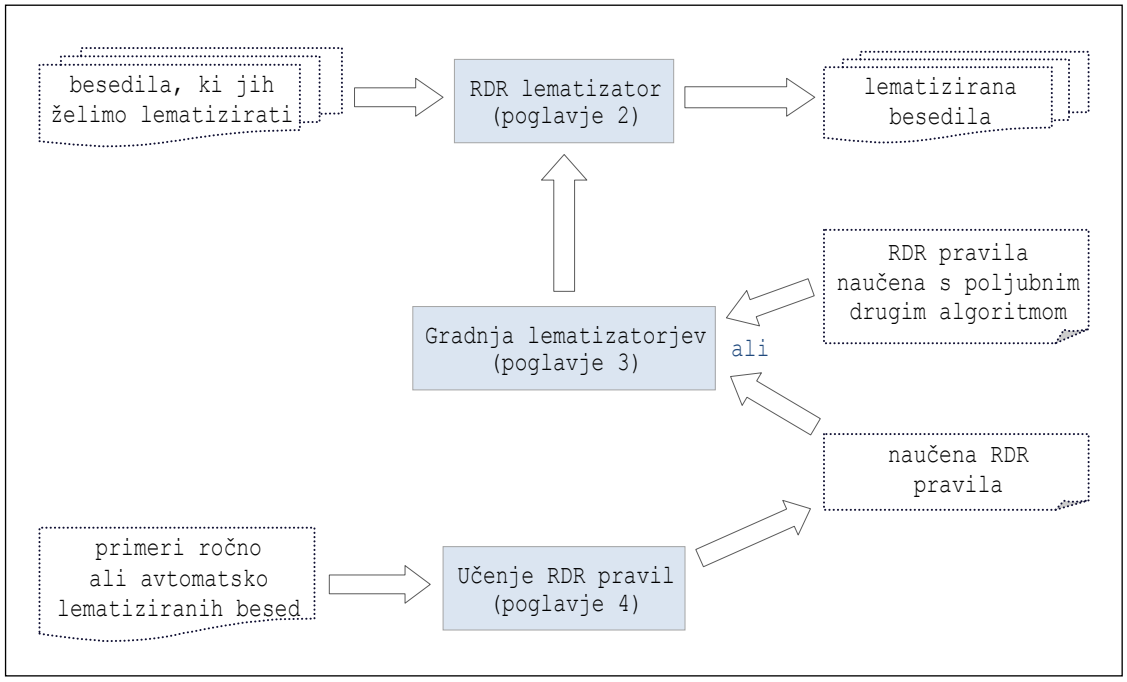


DIAGRAM 1.1: PREGLED INTERAKCIJE MED POSAMEZNIMI SKLOPI NALOGE

2 RDR LEMATIZATOR

Poglavje opisuje osnovni sklop tega diplomskega dela. Sklopa iz poglavij 3 in 4 dejansko podajata le orodja za izdelavo lematizatorja opisanega v tem sklopu. Natančnejšo definicijo uporabe razvitega modula za lematizacijo *Lemmatize*, najdete v prilogi A.

Potreba po učinkovitem lematizatorju tipa RDR se je pojavila kmalu po nastanku članka [12]. Avtorji v članku razmeroma zadovoljivo rešijo problem učenja RDR pravil iz primerov. Klub temu, da je bil lematizator izdelan, pa se avtorji niso poglobljeno ukvarjali z zadnjim korakom tj. uporabo lematizatorja. Za te namene sta bili uporabljeni dve ad-hoc metodi za lematizacijo, s katerima se je dalo pokazati lastnosti učnega algoritma (točnost, standardno odstopanje, …):

- V poglavju (1.3 Metodologija RDR, 1.4 RDR v domeni lematizacije) je razloženo, na kakšen način učni algoritem pridobiva novo znanje iz primerov. Z isto metodo (koda 1.1) lahko iščemo pravila tudi za nove primere. Ideja je sicer preprosta in zanjo skoraj ne rabimo dodatnega truda. Ima pa težavo z implementacijo. Ta je že vnaprej obsojena na slabše delovanje, saj je po RDR principu zgrajena struktura drevesa pravil dokaj neoptimalna že v svoji osnovi. Natančnejše je težava razložena v poglavju (2.2 Alternativi), z njeno rešitvijo pa se ukvarja (3.3.5 Optimizacija). Ta metoda se je uporabljala pri sprotnem testiranju lastnosti učnega algoritma.
- Če pa smo želeli rezultat učenja, torej izdelan lematizator, posredovati tretji osebi, je bilo ukvarjanje s celotnim učnim algoritmom zanjo neprijazno. Za ta namen je bil razvit preprost algoritem, ki tekstovno predstavitev drevesa pravil neposredno pretvori v kodo programskega jezika c++. To se elegantno doseže z uporabo stavčnih oblik if-then-else, za katero obstaja naravna predstavitev RDR pravil. Vendar ima tudi ta metoda težave pri implementaciji. Čeprav je hitrejša od zgornje, se tudi tukaj pojavi isti problem s strukturo drevesa. Poleg tega pa traja prevajanje take avtomatsko generirane kode več minut, kar je nesprejemljivo. Zato je tudi nemogoče vključiti tako kodo v nek večji integriran sistem kot npr. Ontogen.

2.1 ZAHTEVE

Pri snovanju smo si postavili naslednje zahteve:

- Ločena zasnova strukture in algoritma, ki jo zna interpretirati.
- Struktura naj omogoča učinkovito implementacijo.
- Lematizator naj omogoča tak način delovanja, ki bo identičen definiciji sprehajanja po RDR drevesih.

Nadalje pa je implementacija postavila še dodatne pogoje, s katerimi smo zagotovili večjo učinkovitost:

- V pomnilniku naj bo struktura vidna kot en sam povezan blok. Standardne predstavitve dreves s kazalci tako odpadejo.
- Velikost strukture naj bo čim manjša.
- Algoritem naj bo glede porabljenih operacij čim bolj optimalen.

Razlogi za te zahteve so naslednji:

- Če je struktura predstavljena kot en sam blok v pomnilniku, to omogoča trivialne operacije nalaganja in shranjevanja teh podatkov. Podatki se predstavijo kot tabela, nad katero sta ti operaciji za naše velikosti podatkov praktično brez zakasnitve. Dostopna je tudi zelo enostavna vdelava strukture kar v sam program. Program se sicer poveča za njeno velikost, vendar pa se čas prevoda in čas odpiranja takega programa skoraj nič ne podaljšata. Rešitev je zelo praktična tudi zato, ker z njo ne smetimo pomnilnika in omogočamo bistveno večjo izrabo sistemskih zmogljivosti.
- Na zgornje razloge se delno veže tudi velikost strukture. Manjša velikost poleg tega pomeni še pridobitev pri zmanjšanju velikosti programa in tudi pri hitrosti izvajanja. Struktura je dejansko tako majhna, da se za večino jezikov (tudi slovenščino) lahko naloži v predpomnilnik večine današnjih procesorjev in tako omogoča resnično hitro izvajanje.
- Optimalnost algoritma seveda prinese povečano hitrost izvajanja.

2.2 ALTERNATIVI

Z naborom zgornjih zahtev smo se lotili izdelave čim boljšega sistema za lematizacijo. Po obdelavi statistike podatkov in dreves ter poglobljenem razmisleku o čim več možnih alternativah sta se izmed vseh izluščili dve.

2.2.1 VARIANTA 1. ZGOŠČEVALNA TABELA

Za prvo varianto je potrebno pravila iz drevesne oblike spremeniti v zgoščeno *(angl. hash)* tabelo. Indeksi pravil so v tabeli, po neki zgoščevalni funkciji, določeni na podlagi pogoja pravila, torej

končnice besed. Algoritem iskanja pravega pravila nato uporablja lastnost RDR, da je za določeno besedo vedno proženo tisto pravilo, katerega pogoj je najbolj specifičen tj. končnica besede je najdaljša. Zatorej je potrebno pri iskanju pravega pravila vedno začeti z najdaljšo možno končnico besede – kar besedo samo. To zakodiramo po enaki zgoščevalni funkciji in preverimo, če se na dobljenem indeksu res nahaja pogoj s to končnico. Če se, potem lahko vrnemo kar najdeno pravilo. V nasprotnem primeru pa iz končnice izbrišemo prvo črko in ponovimo postopek. Ponavljamo dokler ne dobimo pravila s pravim pogojem. Algoritem se zagotovo ustavi, ko iskana končnica ne vsebuje več nobene črke. Takrat vrnemo pravilo iz korena drevesa, ki po definiciji ustreza vsem besedam.

2.2.2 Varianta 2. Popravljeno RDR drevo

Drugi način iskanja temelji na obratnem postopku. Začnemo na koncu besede oz. s prazno končnico in se postavimo v korensko pravilo RDR drevesa. Dokler še najdemo kakšno izjemo, ki ustreza naši besedi, se premikamo čedalje globlje po vozliščih drevesa. To je pravzaprav natanko princip iskanja, po katerem dela učni algoritem RDR (koda 1.1). Kot smo že omenili, ima ta postopek težave s hitro implementacijo zaradi preveč splošne oblike RDR drevesa.

Po definiciji je vedno potrebno iskati izjeme pravila po vrsti od prvega proti zadnjemu, to pa zato, ker lahko med danimi izjemami več njih ustreza trenutni iskani besedi. To je jedro neučinkovitosti algoritma. Ko pridemo v vozlišče drevesa (pravilo), moramo vedno preiskati vse naslednike (izjeme) dokler ne najdemo takega, ki nam ustreza. Četudi tak ne obstaja, moramo vseeno preveriti vse (za boljšo predstavo: pri drevesu generiranem za slovenščino imajo nekatera vozlišča preko 500 naslednikov). Naslednja težava pojavi ravno v tem dejstvu, namreč zelo veliki vejitvi drevesa v začetnih vozliščih. Kako se temu izogniti? Vemo, da se v vsakem koraku po drevesu navzdol končnica iz pogoja podaljša za najmanj en znak. Če bi lahko dosegli, da se končnica vedno podaljša za natančno eno črko in da so te med seboj različne, potem hkrati rešimo oba problema učinkovitosti. Vejanje zmanjšamo na maksimalno toliko, kot je črk v abecedi jezika. V vsakem vozlišču pa še lahko direktno najdemo izjemo za trenutno besedo, če le ta obstaja. Izkaže se, da obstaja postopek (3.3.5 Optimizacija), ki popravi drevo na tak način, da se njegova semantika ne spremeni, vendar dobi drevo zgoraj opisane lepe lastnosti.

Primer 2.1 prikazuje na zgornji način popravljeno RDR drevo iz primera 1.4. Poglavitne spremembe so npr. dodano novo vozlišče s končnico "mo" ter dodana izjema v vozlišče "i". Poleg tega pa lahko opazimo tudi ostale, na prvi pogled kozmetične popravke, ki so posledica algoritma. Zgled tega je ureditev vozlišč po dodani črki.

PRIMER 2.1: POPRAVLJENO RDR DREVO

```
1      RULE:( suffix("") transform(""-->"") except(3) ); {:
1.1     |---> RULE:( suffix("i") transform("i"-->"o") except(3) ); {:
1.1.1   |       |---> RULE:( suffix("ni") transform("ni"-->"ti") );
1.1.2   |       |---> RULE:( suffix("ti") transform(""-->"") );
1.1.3   |       `---> RULE:( suffix("ši") transform("ši"-->"sati") ); :}
        |
1.2     |---> RULE:( suffix("l") transform("l"-->"ti") );
1.3     `---> RULE:( suffix("mo") transform(""-->"") except(2) ); {:
1.3.1         |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
1.3.2         `---> RULE:( suffix("šimo") transform("šimo"-->"sati") ); :}
               :}
```

V nadaljevanju tega poglavja obravnavamo tako popravljeno drevo. Naše drevo zdaj prilagodimo na naslednji način. V vsakem notranjem vozlišču dodamo majhno zgoščevalno tabelo, ki pove, na kateri poziciji se morebiti nahaja izjema za dano besedo. Tako preverimo vsakič zgolj en pogoj, ne glede na to, koliko izjem ima pravilo.

2.3 IMPLEMENTACIJSKA SHEMA

Odločili smo se za realizacijo variante 2 opisane v poglavju 2.2.2. Razlogi so naslednji:

- Za hitro izvedbo zgoščevalne tabele v prvi varianti bi težko našli popolno zgoščevalno funkcijo. Želeli bi si, da ta funkcija idealno razdeli pravila tako, da nobena dva pravila nimata enakega mesta v tabeli. Poleg tega pa bi radi čim boljše razmerje med polnimi in praznimi celicami tabele.
- Že sam izračun zgoščevalne funkcije bi verjetno upočasnil prvo metodo. Za izračun bi morala funkcija konsolidirati vse črke besede, medtem ko slednja izračuna zelo enostavno funkcijo vedno le na podlagi ene črke.
- Pri prvi metodi vedno napredujemo za natanko eno črko po besedi naprej. Pri drugi metodi pa v povprečju napredujemo po besedi nazaj za več kot eno črko.

2.3.1 NOTACIJA

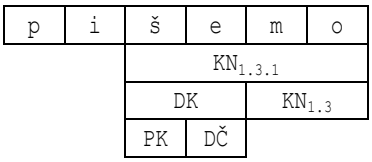
Za potrebe naslednjih razdelkov je potrebno definirati nekaj pojmov, s katerimi je mogoče bistveno enostavneje opisati strukturo in algoritem. Pravila se sklicujejo na primer 2.1 in so oštevilčena glede na pripadajoče številke na začetku vrstic.

- **Končnica (KN)** je celotna končnica pravila. Vse besede, ki prožijo to pravilo, se morajo končati s to končnico.
  - *KN(pravila 1.2) = 'l'*
  - *KN(pravila 1.3) = 'mo'*
  - *KN(pravilo 1.3.1) = 'šemo'*

- **Dodana končnica (DK)** je tisti del končnice, ki je bil dodan v zadnjem vozlišču glede na nadrejeno vozlišče. KN trenutnega vozlišča je torej stik DK vseh vozlišč na poti od korena do tega vozlišča *[KNi = DKi · DKi-1 · … · DK1 · DK0].*
  - *DK(pravila 1.2) = 'l'*
  - *DK(pravila 1.3) = 'mo'*
  - *DK(pravila 1.3.1) = 'še'*
- **Dodana črka (DČ)** je zadnja črka dodane končnice. Kadar je dolžina dodane končnice enaka ena, takrat je DČ kar enaka DK. Dodane črke izjem istega pravila morajo biti različne, saj se upoštevajoč DČ išče izjeme pravila.
  - DČ(pravila 1.2) = 'l'
  - DČ(pravila 1.3) = 'o'
  - DČ(pravila 1.3.1) = 'e'
- **Podaljšana končnica (PK)** je prvi del dodane končnice brez zadnje črke. Stik PK in DČ tako vedno predstavlja DK *[DK = PK · DČ]*. Čeprav smo v (2.2.2 Varianta 2. Popravljeno RDR drevo) napisali, da se KN v optimiziranem drevesu v vsakem vejanju podaljša za natančno eno črko, temu ni čisto tako. Včasih lahko dve ali več vozlišč združimo in tako zelo zmanjšamo velikost drevesa. Ravno zaradi tega lahko pride do pojava ko *[PK ≠ '']*. Več o tem je razloženo v (3.3.5 Optimizacija).
  - *PK(pravila 1.2) = ''*
  - *PK(pravila 1.3) = 'm'*
  - *PK(pravila 1.3.1) = 'š'*

Bolj nazorno so zgornji pojmi prikazani v primeru 2.2.

PRIMER 2.2: DEFINICIJE KONČNIC NA PRAVILU 1.3.1



2.3.2 STRUKTURA

Sledi definicija strukture za predstavitev optimiziranih RDR dreves. Poznamo 4 različne tipe vozlišč drevesa:

- Tip 0: Vozlišče tipa 0 je pravzaprav pravilo oz. transformacija, kako popraviti končnico besede, da dobimo lemo. Podatki, ki so prisotni v tem vozlišču, so:
  - dolžina končnice besede, ki jo moramo odrezati
  - dolžina končnice leme, ki jo moramo prilepiti obrezani besedi
  - končnica leme, ki jo moramo prilepiti

- Tip 1: To vozlišče nima naslednikov, vendar kljub temu še ni končno. Vsebuje namreč podaljšano končnico pogoja. Podatki so:
  - kazalec na tip 0, ki ga vrnemo, če je pogoj podaljšane končnice izpolnjen
  - dolžina podaljšane končnice
  - podaljšana končnica
- Tip 2: Pri tipu 2 in 3 imamo opravka z notranjimi vozlišči, torej takimi, ki imajo naslednike. Podatki:
  - kazalec na tip 0, ki ga vrnemo, če beseda ne ustreza nobeni izjemi tega vozlišča
  - definicija zgoščevalne funkcije
  - zgoščevalna tabela naslednikov (izjem) tega vozlišča
- Tip 3: Kot tip 2 le, da vsebuje še pogoj podaljšane končnice:
  - kazalec na tip 0, ki ga vrnemo, če je pogoj podaljšane končnice izpolnjen in če beseda ne ustreza nobeni izjemi tega vozlišča
  - dolžina podaljšane končnice
  - podaljšana končnica
  - definicija zgoščevalne funkcije
  - zgoščevalna tabela naslednikov tega vozlišča

2.3.3 Algoritem

Algoritem je v osnovi zelo enostaven. Njegova naloga je sprehod po drevesu do tistega vozlišča, ki ga proži določena beseda. Ko najde tako vozlišče, z njegovim pravilom le še transformira besedo in vrne njeno lemo. Najpomembnejši nalogi med iskanjem proženega pravila sta preverjanje podaljšanih končnic in iskanje izjem trenutnega pravila. Iskanje se ustavi, ko pogoj podaljšane končnice ni izpolnjen, ali ko trenutno pravilo nima izjem. V prvem primeru je trenutno vozlišče napačno, zato za transformacijo uporabimo pravilo nadrejenega vozlišča. V slednjem pa se trenutno vozlišče proži, zato uporabimo njegovo pravilo.

2.4 Implementacija

V tem razdelku sta bolj podrobno razloženi implementaciji strukture in algoritma. Na implementacijo ima poseben vpliv zahteva, da naj drevo predstavlja povezan blok v pomnilniku. Brez te zahteve bi bila realizacija z objekti in kazalci sicer preprostejša, a bi izgubili nekaj zgoraj naštetih lepih lastnosti.

Pretvorba objekta, v našem primeru RDR drevesa, v tok zaporednih podatkovnih enot se imenuje serializacija. Obratni postopek pretvorbe toka podatkov v objekte pa se imenuje deserializacija. Z deserializacijo se v tem delu eksplicitno ne ukvarjamo. Algoritem za lematizacijo se namreč sprehaja po serializiranem toku podatkov in ga sproti interpretira. S pravilno implementacijo je ta način še hitrejši kot iskanje po deserializiranem drevesu. Implicitno deserializacijo izvajajo tudi druge, pomožne metode, ki interpretirajo drevo, na primer izpis drevesa.

2.4.1 SERIALIZACIJA

Serializacija vsakega objekta mora poleg samega kodiranja v podatkovni tok upoštevati tudi možnost branja objekta iz tega toka oz. v našem primeru interpretacije objekta v samem toku. Kodiranje mora biti nedvoumno in za vsak prebrani podatek mora biti določljivo, kateremu objektu in delu objekta pripada. Kodiranje tabele mora tako na primer vsebovati tako zapis o njeni dolžini kot tudi o tipu podatkov v njej. Informacija o tipu podatka je pogosto podana implicitno v samem algoritmu kodiranja oz. interpretacije.

Oblika toka podatkov, za katerega smo se odločili, je niz bajtov *(angl. byte).* Osnovna enota bajt nudi najboljši kompromis med hitrostjo izvajanja in velikostjo strukture, čeprav so določeni podatki manjši, npr. podatek o tipu zavzema le 4 bite. Vendar bi za branje ostalih podatkov na lokacijah v pomnilniku, ki niso poravnane z bajtom, porabili dosti več časa kot ga sedaj. Tako tudi za tip porabimo kar en bajt. V algoritmu je niz bajtov realiziran kot tabela, saj je tako dostop najbolj intuitiven.

Pri kompleksnejših strukturah, kakršno je tudi drevo, moramo določiti, kako bomo kodirali kazalce. Kazalci so mišljeni kot povezave med nadrejenimi in podrejenimi vozlišči. Definirali smo jih podobno kot so naslovi v pomnilniku. Ker pa imamo celotno drevo zdaj stisnjeno v eno tabelo, ne rabimo več dostopat do celotnega pomnilnika. Novi kazalci so torej naslovi celic tabele. Npr. 0 je kazalec na prvo celico, 1 na drugo, 2 na tretjo itd. do konca tabele. Naša tabela je v normalnih pogojih bistveno manjša od celotnega naslovnega prostora, zato lahko te kazalce skrajšamo iz 4B na 3B. Z naslovi dolgimi 3B lahko dosežemo velikost tabele 2^{3*8}B = 2^{24}B = 16.384KB. Če upoštevamo povprečno velikost vozlišča (podatki za slovenščino) cca. 15B, vidimo, da lahko s 3B naslovi zakodiramo drevesa s približno milijon vozlišči. To pa je v primerjavi z dejanskim drevesom kar 30 krat več. Na ta način lahko ob cca. 20% prihranku pri velikosti tabele zakodiramo dokaj velika drevesa. Kljub temu pa se lahko zgodi, da je drevo večje. Problem se pojavi zelo redko in še to pri jezikih, katerim RDR lematizacija naravno ne ustreza in se drevo napihne. Zato dejanski algoritem za gradnjo (3.3.6 Serializacija) najprej zgradi tabelo z naslovi velikosti 3B, preveri njeno velikost in če je ta prevelika, spremeni velikost naslovov na 4B ter ponovno zgradi tabelo. Zaradi enostavnosti bomo v nadaljevanju za kazalce uporabljali kar velikost 3B.

TABELA 2.1: DEFINICIJA SERIALIZACIJE VOZLIŠČ

ZA VSAK TIP PODATKA IMAMO PODAN NJEGOV NAZIV, DOLŽINO V BAJTIH TER ODMIK PODATKA OD ZAČETKA VOZLIŠČA. ZA NEKATERA POLJA SE UPORABLJA NASLEDNJA NOTACIJA:

- D(X) JE DOLŽINA PODATKA X V B,
- N(X) JE NASLOV VOZLIŠČA X V TABELI,
- P JE PRAVILO, TOREJ VOZLIŠČE TIPA 0, KI PRIPADA TRENUTNEMU VOZLIŠČU,
- KN, PK IN DČ SO DEFINIRANI V (2.3.1 NOTACIJA)

tip 0

podatek:	tip	d(KN$_{beseda}$)	d(KN$_{lema}$)	KN$_{lema}$	
velikost:	1B	1B	1B	d(KN$_{lema}$)	
odmik:	0B	1B	2B	3B	3B+d(KN$_{lema}$)

tip 1

podatek:	tip	n(p)	d(PK)	PK	
velikost:	1B	3B(4B)	1B	d(PK)	
odmik:	0B	1B	4B	5B	5B+d(PK)

tip 2

podatek:	tip	n(p)	d(izjeme)	izjeme[]	
velikost:	1B	3B(4B)	1B	d(izjeme)*4B	
odmik:	0B	1B	4B	5B	5B+d(izjeme)*4B

tip 3

podatek:	tip	n(p)	d(PK)	PK	d(izjeme)	izjeme[]	
velikost:	1B	3B(4B)	1B	d(PK)	1B	d(izjeme)*4B	
odmik:	0B	1B	4B	5B	5B+d(PK)	6B+d(PK)	6B+d(PK)+d(izjeme)*4B

izjeme[]

podatek:	DČ	n(tip 0-3)	DČ	n(…)	…	…	DČ	n(…)	DČ	n(tip 0-3)	
velikost:	1B	3B(4B)	…	…	…	…	…	…	1B	3B (4B)	
odmik:	0B	1B	…	…	…	…	…	…	(d(izjeme)-1)*4B	(d(izjeme)-1)*4B+1B	d(izjeme)*4B

Problem serializacije lahko ločimo na dva podproblema:

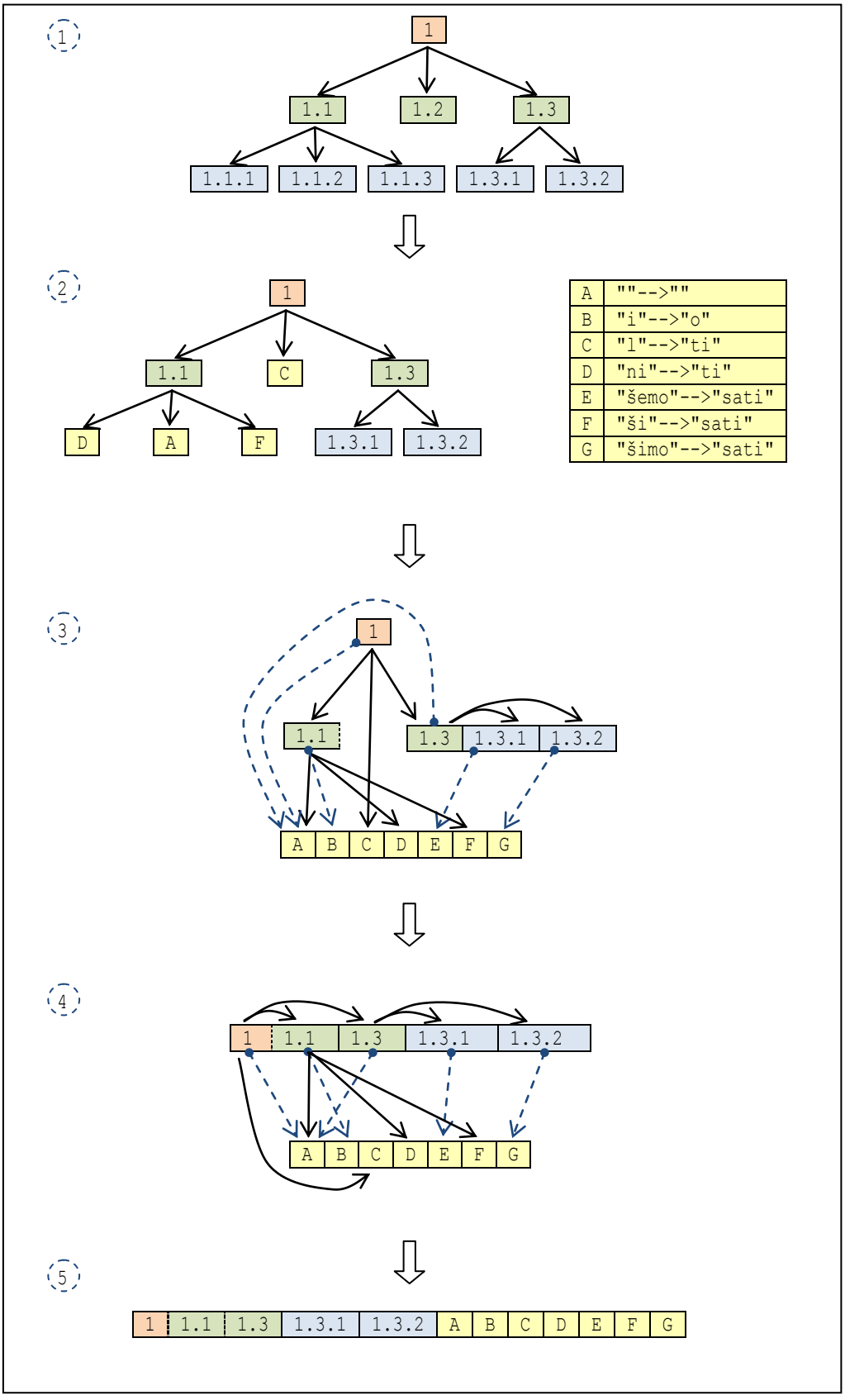
- Serializacija vozlišč določi, kako se zakodirajo podatki samega vozlišča. Katere podatke potrebujemo za vsak tip vozlišča, je v grobem nakazano že v prejšnjem podpoglavju. Kako se vozlišča dejansko serializirajo, podaja tabela 2.1. Vidimo, da je pri vseh štirih tipih prvi podatek prav tip vozlišča. To je edini način, da lahko zagotovimo enolično prepoznavanje vozlišča. Vedno, ko se nahajamo v vozlišču na odmiku 0, lahko preverimo, katero je to vozlišče. Naslednji podatki pa se razlikujejo glede namembnosti. Tip 0 podaja tri podatke, s katerimi lahko transformiramo besedo v njeno lemo. Ostali trije pa imajo kot prvi podatek ravno naslov pravila tipa 0. Tip 1 ima poleg tega še podatek o podaljšani končnici, 2 zgoščevalno tabelo izjem pravila, 3 pa oboje. Tabela izjem je enaka za 2 in 3 in je prikazana ločeno. Njena zgradba je preprosta: vsebuje množico parov <dodana črka, naslov pravila>. Zadnji podatek v vrstici odmik je položaj naslednjega vozlišča glede na trenutnega, lahko ga pa interpretiramo tudi kot dolžino celotnega vozlišča izraženo v bajtih.
- Serializacija drevesa definira, na kašen način je zakodirana hierarhija vozlišč. Odločili smo se za zaporedje kodiranja: "najprej v globino" *(angl. depth-first)*. Obiskovanje vozlišč po tem principu je enako kot pri znanem algoritmu iskanja v globino. Vendar je ta postopek nekoliko prilagojen naši problemski domeni. Vozlišče tipa 0 je pravzaprav pravilo o

transformaciji besede v lemo. Vsako vozlišče drugega tipa mora imeti kazalec na eno vozlišče tipa 0. Seznam pravil zgradimo enostavno tako, da obiščemo vsa vozlišča in če naletimo na pravilo, ki ga še ni v seznamu, ga dodamo. Seznam je v naši konkretni implementaciji urejen po abecednem vrstnem redu in ga po koncu serializacije prilepimo drevesu. Tak način kodiranja vozlišč tipa 0 se kljub dodatnemu nivoju v drevesu izplača zaradi večkratnega ponavljanja enakih pravil. Postopek kodiranja drevesa je prikazan v primeru 2.3 in si zasluži natančnejšo obrazložitev:

- V prvem koraku vidimo originalno RDR drevo iz primera 2.1.
- Najprej generiramo seznam pravil. Vsa tista vozlišča, katera nosijo zgolj informacijo o transformaciji besede v lemo (1.1.1, 1.1.2, 1.1.3, 1.2), lahko iz drevesa odstranimo in njihovim nadrejenim vozliščem spremenimo kazalec na zapis v seznamu pravil.
- V korakih 3 in 4 nato postopoma od spodaj navzgor združujemo vozlišča tako, da otroke vedno zaporedoma enega za drugim prilepimo staršem.
- Končni rezultat koraka 5 je tabela, ki predstavlja izhodiščno drevo. Seveda je tudi tukaj prisotna informacija o kazalcih na podrejena vozlišča, vendar zaradi nazornosti puščice niso narisane.
- Na diagramu predstavljajo neprekinjene črne puščice tiste kazalce, ki izvirajo iz zgoščevalnih tabel, prekinjene modre pa tiste, ki kažejo na privzeto pravilo vozlišča (označeno z n(p) v tabeli 2.1).

Ob tem je potrebno bolje doreči tudi način kodiranja zgoščevalne tabele. Zgoščevalna tabela je v bistvu tabela kazalcev na izjeme trenutnega vozlišča. Ker je bila naša želja, da imamo do izjeme direktni dostop, brez iskanja po celotnem seznamu izjem, mora biti ta tabela zgrajena nekoliko kompleksnejše. Ključ za dostop do izjeme je DČ (dodana črka). Naj bo dolžina zgoščevalne tabele N. Ta podatek je zapisan v bajtu tik pred začetkom tabele. Indeks v tabeli, kjer se morda nahaja izjema za določeno DČ, dobimo kot ostanek pri deljenju DČ z N. (Index = DČ % N). Vsakemu indeksu pa seveda ustreza več črk. Če je N npr. 10, potem so na indeksu 7 v tabeli lahko izjeme za DČ a(97), k(107) ali u(117). Vsak zapis tabele tako vsebuje tudi DČ, kateri ustreza izjema. To shranjeno DČ moramo vedno preverjati z našo DČ. Če sta ti enaki, smo zares našli izjemo sicer pa ne. Drugi podatek je naslov vozlišča, ki ustreza tej izjemi. Tabela je, kot rečeno, seznam parov <DČ, naslov izjeme>, njena dolžina pa je hkrati tudi podatek za funkcijo iskanja pravega para.

PRIMER 2.3: KONCEPT SERIALIZACIJE DREVESNE HIERARHIJE PRIKAZAN NA DREVESU IZ PRIMERA 2.1



V kolikor so ostale še kakšne nejasnosti v zvezi z našim postopkom kodiranja drevesa, navajamo v primeru 2.4 zgled, kako se serializira drevo iz primera 2.1. Ta zgled je logično nadaljevanje primera 2.3 s tem, da tukaj uporabimo tudi pravila za kodiranje posameznih vozlišč. Zgornja vrstica vsakega podatka tako predstavlja prav referenco na vozlišča iz slednjega primera, spodnja pa dejanske podatke.

Lokacijo posameznih podatkov v tabeli izračunamo tako, da seštejemo naslovno vrstico in naslovni stolpec želene celice. Barvna shema je povzeta iz tabele 2.1 in s primerjavo lahko hitro identificiramo, za katere podatke gre.

PRIMER 2.4: ZGLED SERIALIZACIJE DREVESA

NASLOVI OZ. KAZALCI NA DRUGE CELICE TABELE SO ZARADI BOLJŠE PREGLEDNOSTI OZNAČENI Z @. ZNAK / PA POMENI PRAZNA POLJA V ZGOŠČEVALNI TABELI, DEJANSKO PA SO V TEH POLJIH ZAPISANE NIČLE.

naslovi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1(@0)																					1.1(@21)										
	2	@77		4	l		@84		i		@21		/		/		o		@46		2		@80		5	n		@89		t		
32														1.3(@46)																		
	@77	/		/		/				š		@101		3		@77		1	m	3	i		@71		/		/			e		@65	
64		1.3.1(@65)					1.3.2(@71)							A(@77)			B(@80)				C(@84)				D(@89)							
		1		@94		1	š	1		@108		1	š	0	0	0	0	1	1	o	0	1	2	ti	0	2	2	ti	0	4		
96	E(@94)					F(@101)						G(@108)																				
	4	sati				0	2	4	sati			0	4	4	sati																	

Pojavna oblika serializiranega drevesa v izvorni kodi programa je prikazana v kodi 2.1. Čeprav v splošnem drevesa niso ravno velika (cca 0.5MB), pa zapis v desetiški oz. šestnajstiški obliki zavzame približno 2-3x toliko prostora. Kadar želimo kodo strukture vključiti v program, se pojavi povečanje zgolj v velikosti izvornih datotek. Ker je šestnajstiški zapis vseeno malce optimalnejši, se v programu uporablja izključno ta. Zapis z bazo 10 je tukaj naveden kot korak prehoda iz primera 2.4 v bazo 16.

Kljub naštetim dobrim lastnostim serializacije pa se v tem primeru vidi tudi največja pomanjkljivost. Za človeka je struktura praktično neberljiva. Če bi želeli tako drevo analizirati, ga je potrebno nujno prebrati s programom in izvoziti v prijaznejši obliki. Kadar pride v tej tabeli do kakšne napake, jo je praktično nemogoče najti, saj take okvarjene strukture ne moremo interpretirati programsko.

Celotna struktura zavzame 115B, vendar se velikost vedno zaokroži navzgor na večkratnik 8B. V tabeli se to realizira tako, da se na koncu doda potrebno število (od 0 do 7) podatkov velikosti enega bajta in se jih inicializra na 0. Z zaokrožitvijo se poenostavi izvoz in uporaba drevesa v obliki kode c++. Dejanska velikost našega drevesa tako znese 120B.

KODA 2.1: SERIALIZIRANO DREVO V C++ NOTACIJI TER BAZAH 10 IN 16[1]

```
1  {  2, 77,  0,  0,  4,'l', 84,  0,      0,'i', 21,  0,  0,  0,  0,  0,
2     0,'o', 46,  0,  0,  2, 80,  0,      0,  5,'n', 89,  0,  0,'t', 77,
3     0,  0,  0,  0,  0,  0,  0,  0,      0,  0,'š',101,  0,  0,  3, 77,
4     0,  0,  1,'m',  3,'i', 71,  0,      0,  0,  0,  0,  0,'e', 65,  0,
5     0,  1, 94,  0,  0,  1,'š',  1,    108,  0,  0,  1,'š',  0,  0,  0,
6     0,  1,  1,'o',  0,  1,  2,'t',    'i',  0,  2,  2,'t','i',  0,  4,
7     4,'s','a','t','i',  0,  2,  4,    's','a','t','i',  0,  4,  4,'s',
8   'a','t','i',  0,  0,  0,  0,  0}
```

```
1  {0x00546c0400004d02, 0x0000000000156900,
2   0x00500200002e6f00, 0x4d740000596e0500,
3   0x0000000000000000, 0x4d030000659a0000,
4   0x004769036d010000, 0x0041650000000000,
5   0x019a0100005e0100, 0x0000009a0100006c,
6   0x740201006f010100, 0x0400697402020069,
7   0x0402006974617304, 0x7304040069746173,
8   0x0000000000697461}
```

2.4.2 LEMATIZACIJSKI ALGORITEM

Tudi v naši problemski domeni se izkaže, da ob pravilnem načrtovanju podatkov postane algoritem enostaven, njegova koda pa kratka. Mislimo, da nam je v tem delu naloge to uspelo. Čeprav so podatki stisnjeni (serializirani), se algoritem elegantno in učinkovito sprehaja po njih ter išče pravila za lematizacijo.

Algoritem je podan v kodi 2.2. Vhod v lematizator sta poljubna morfološka oblika besede ter tabela RDR drevesa, izhod pa lematizirana beseda. Potek algoritma je sledeč:

- 4-7: Izhodiščno oz. korensko vozlišče se vedno nahaja na začetku tabele (naslov 0). Inicializirati pa moramo tudi tip vozlišča in kazalec na črko *LookChar*, ki kaže na konec besede Word.
- 9-34: Po drevesu se sprehajamo tako dolgo, dokler ne končamo v vozlišču tipa 0.
  - 10-19: V primeru tipov 1 in 3 je potrebno preveriti pogoj ujemanja PK. Možni so 3 različni izhodi. V zanki ostanemo, če je tip enak 3 in pogoj PK izpolnjen.
  - 21-22:Po besedi se premaknemo za eno črko nazaj. Če je to prvi obhod zanke, potem po končani vr. 22 gledamo na zadnjo črko besede.
  - 24-34: V kolikor se nahajamo v notranjih vozliščih (takih, ki imajo izjeme, tip 2 in 3), potem poskušamo najti izjemo za dano besedo.

[1] V šestnajstiškem zapisu se bajti znotraj besede berejo od desne proti levi po dve števki naenkrat.
[2] Pravilo z večjo zaporedno številko je namreč v originalnem drevesu ležalo za tistim z nižjo, zato algoritem nikoli ne bi mogel dostopati do njega.
[3] Podobno kot v opombi 2. Takšno pravilo je nedostopno.
[4] Podrejeni je bil v originalnem drevesu naveden pred nadrejenim. V primeru lematizacije besede, ki ustreza

- 25-26: Izračunamo mesto v zgoščevalni tabeli, kjer se lahko nahaja izjema glede na črko, ki jo gledamo.
- 28-30: Črka, ki je zapisana v tabeli, je enaka kot trenutna gledana. Torej je izjema pravilna. Iz tabele preberemo še naslov novega vozlišča ter se premaknemo vanj.
- 31-32: Izjema ne ustreza trenutni opazovani črki, zato uporabimo lematizacijsko pravilo trenutnega vozlišča. Premaknemo se v to pravilo. S tem zaključimo tudi izvajanje *while* zanke.

- 37-40: Po dobljenem pravilu lematiziramo besedo in jo vrnemo iz funkcije.

KODA 2.2: PSEVDOKODA ALGORITMA LEMATIZATORJA

```
1  //word ... beseda, ki jo želimo lematizirati
2  //tree ... tabela, ki vsebuje serializirano drevo
3  Lemmatize(tree, word)
4      address = 0              //naslov trenutnega vozlišča
5      parentAddr = 0           //naslov nadrejenega vozlišča
6      lookChar = Length(word)  //katero črko končnice besede word trenutno obravnavamo
7      type = tree[address]     //tip vozlišča (0,1,2 ali 3)
8
9      while(type != 0)
10         if (type == 1 or 3)  //po potrebi preveri pogoj podaljšane končnice
11             //pogoj PK ni izpolnjen, vrni se na nadrejeno vozlišče
12             if (tree[address + Offset(PK)] != PK(word))
13                 address = parentAddr
14                 break
15             //pogoj PK je izpolnjen, ampak smo v vozlišču tipa 1, ki je končno
16             if (type == 1)
17                 break
18             //sicer premaknemo kazalec na črko toliko nazaj, kot je bila dolžina PK
19             lookChar -= tree[address + offset(PKLenght)]
20
21         lookChar--               //premaknemo trenutno opazovano črko za 1 nazaj
22         if (lookChar<0) break    //konec, če smo šli prek začetka besede
23
24         if(type == 2 or 3)       //vozlišče ima izjeme, zato jih poskušamo najti
25             hashIndex = word[lookChar] % tree[address + Offset(hashLength)]
26             exceptionAddr = address + Offset(hashTable) + 4*HashIndex
27
28             if (word[lookChar] == tree[exceptionAddr]) //izjema dejansko obstaja
29                 parentAddr = address             //shranimo podatek o nadrejenem vozl.
30                 address = tree[exceptionAddr + 1] //premaknemo se v izjemo
31             else                                 //v okviru dane bes. izjema ne obstaja
32                 address = tree[address + 1]      //premaknemo se na tip 0 tega vozlišča
33
34             type = tree[address]      //nahajamo se v novem vozlišču zato posodobimo tip
35
36     //nahajamo se v vozlišču tipa 0, zato lematiziramo besedo
37     lemma = DeleleSuffix(word, tree[address + 1])
38     lemma = AppendSuffix(Lemma, tree[address + 3])
39
40     return lemma //vrnemo lematizirano besedo
```

2.4.3 Razred

V izvorni kodi se razred za lematizacijo imenuje RdrLemmatizer. Deklaracija tega razreda je prikazana v kodi 2.3. Poleg že opisane funkcije Lemmatize vsebuje razred še naslednje pomembne elemente:

- Spremenljivka abData je tabela bajtov serializiranega drevesa.
- Spremenljivka iDataLen predstavlja dolžino tabele abData.
- Konstruktor RdrLemmatizer: če ga uporabimo brez parametrov, nastavi privzeto drevo, ki pa je lahko polno ali prazno odvisno od načina generiranja razreda. Sicer pa mu preko parametra podamo tabelo drevesa.
- Funkcija ToString v izhodni tok *(angl. stream)* zapiše predstavitev drevesa, ki je berljiva za ljudi. Hkrati pa je drevo tudi po standardu iz (3.2 Določitev vhodne datoteke) in ga je možno ponovno uvoziti v program.
- Funkcija ToStringHex v izhodni tok zapiše predstavitev drevesa, ki je primerna za vključitev v izvorno kodo programa (koda 2.1, baza 16).
- Funkcija SaveBinary shrani serializirano drevo v tok bajt za bajtom.
- Funkcija LoadBinary naloži shranjeno serializirano drevo iz tok bajtov v spremenljivko abData.

Koda 2.3: Deklaracija razreda RdrLemmatizer

```
1  class RdrLemmatizer{
2  private:
3      byte *abData;
4      int iDataLen;
5
6  public:
7      RdrLemmatizer(byte *abData, int iDataLen) ;
8      RdrLemmatizer();
9      ~RdrLemmatizer();
10
11     int SizeOfTree() const;
12
13     char *Lemmatize(const char *acWord, char *acOutBuffer = NULL) const;
14
15     void ToStringHex(ostream &os) const;
16     void SaveBinary(ostream &os) const;
17     void LoadBinary(istream &is);
18
19     void ToString(ostream &os = cout, dword iStartAddr = -1, int iDepth = 0,
20         char *acParSufx = "", char *acParDev = "", char cNewChar=NULL) const;
21 };
```

3 GRADNJA LEMATIZATORJEV

Poglavje predstavlja drugi sklop diplomske naloge. V njem obravnavamo postopek, kako iz poljubnega RDR drevesa izdelamo lematizator opisan v prejšnjem poglavju. Iz diagrama 1.1 je razvidno, na kakšen način to poglavje povezuje ostali dve. Vhod v program je datoteka človeku berljivih RDR pravil v dogovorjeni notaciji, izhod pa serializirana struktura lematizatorja v binarni datoteki oz. skupaj z algoritmom v obliki izvornih datotek lematizatorja. Opis uporabe modula za gradnjo lematizatorjev *LemBuild* najdete v prilogi A.

3.1 ZAHTEVE

Za izvedbo naloge smo zaradi želje po čim večji splošnosti rešitve sprejeli naslednje cilje:

- Lematizator naj sledi zasnovi iz prejšnjega poglavja. Izhod tega programa je torej serializirano drevo, ki za delovanje potrebuje zgornji lematizator.
- Delovanje mora biti povsem enako, kot če bi lematizirali z izhodiščnim RDR drevesom pravil.
- Vhodno drevo mora biti sicer zapisano v pravilni notaciji, vendar je vsebinsko lahko poljubno. Drevesa so namreč lahko (kljub pravilni notaciji) napačno generirana, npr. vsebujejo nedosegljiva vozlišča. Želimo omogočiti tudi obdelavo tistih dreves, katerih učnega algoritma ne poznamo. S tem omogočimo tudi ročno generirana drevesa, ki lahko vsebujejo nekatere nedoslednosti. Program naj v takem primeru drevo popravi tako, da bo rezultat lematizacije identičen rezultatu vhodnega drevesa.
- Natančna specifikacija vhodne datoteke: notacija naj bo čim svobodnejša vendar dovolj striktna, da lahko program zazna leksikalne napake. V primeru napake naj uporabniku poskuša čim bolj točno opredeliti mesto in izvor napake.

Iz vsebine naloge in teh ciljev sledi, da želimo pravzaprav narediti prevajalnik *(angl.: compiler)*. V splošnem prevajalnik sprejme kot vhod zapis programa v nekem programskem jeziku in izdela njegovo izvršilno kodo v jeziku stroja, za katerega prevaja. V našem primeru je zapis programa kar

zapis RDR drevesa, programski jezik pa notacija, ki smo jo definirali. Izhod je koda (serializirano drevo), ki jo razume naš stroj tj. algoritem lematizatorja. Zaradi tega smo določili še dodatni zahtevi:

- uporaba standardne blok sheme prevajalnika (leksikalna, sintaksna in semantična analiza, optimizacija, …)
- izdelava prvih dveh stopenj (leksikalna ter sintaksna analiza) s standardnimi orodji. S tem poenostavimo kodo in povečamo zanesljivost teh stopenj.

3.2 Določitev vhodne datoteke

Naša želja je bila omogočiti uporabnikom programa čim svobodnejše oblikovanje vhodne datoteke. Nekateri želijo imeti npr. čim bolj nazorno drevo z vmesnimi praznimi vrsticami ter črtami za nakazovanje hierarhije, drugim pa mogoče več pomeni velikost datoteke in želijo čim kompaktnejši zapis. Oblika datoteke je tukaj podana le opisno, podrobnejšo definicijo pa podajata poglavji (3.3.1 Leksikalna analiza) ter (3.3.2 Sintaksna analiza).

Osnovna enota vsake datoteke je eno pravilo *(angl. rule)*. Poleg pravila v datoteki ločimo še oznaki za začetek in konec seznama pravil ter komentarje. Pravila se vedno začnejo s ključno besedo *"rule:"*. Sistem je zasnovan tako, da so vse ključne besede neodvisne od velikosti znakov, zato lahko napišemo tudi *"Rule:"*, *"RulE:"* ali morda *"RULE:"*, kakor nam je ljubše. Posamezne elemente pravila imenujemo lastnosti. Vsako pravilo vsebuje vsaj dve obvezni lastnosti, končnico in transformacijo. Poleg teh dveh pa lahko uporabimo še identifikacijsko oznako pravila in/ali število izjem. Pravilo se zaključi s podpičjem, koncem vrstice ali koncem datoteke. Nato program do naslednje ključne besede za pravilo sprejema komentarje. Komentarje sestavljajo zaporedja poljubnih znakov. Če znotraj komentarja sistem zazna oznako za začetek seznama pravil, se začne graditi seznam izjem zadnjega sprejetega pravila. Ta seznam se mora zaključiti z znakom za konec seznama.

Primer 3.1 prikazuje le nekaj oblik notacij, ki so možne glede na spodnjo definicijo leksikalnega analizatorja (*angl. lexer*, podano v poglavju 3.3.1) in sintaksnega analizatorja (*angl. parser*, podano v poglavju 3.3.2). Prvi trije zgledi izpisov so generirani z modulom *LemLearn* (priloga A), ki ponuja tudi nekaj drugih vnaprej pripravljenih formatov. V kolikor želimo popolnoma drugačno obliko drevesa, lahko vedno uporabimo pristop uporabljen v četrtem zgledu. V okviru ene datoteke lahko uporabljamo poljubno število različnih slogov.

PRIMER 3.1: PRIMERI RAZLIČNIH NOTACIJ

```
1  1      `---> RULE:( suffix("") transform(""-->"") except(2) ); {:
   1.1           |---> RULE:( suffix("i") transform("i"-->"o") except(2) ); {:
   1.1.1         |      |---> RULE:( suffix("ni") transform("ni"-->"ti") );
   1.1.2         |      `---> RULE:( suffix("ti") transform(""-->"") ); :}
                 |
   1.2           `---> RULE:( suffix("l") transform("l"-->"ti") ); :}
```

```
2  1      rule: i'' t''->''; {:
   1.1          rule: i'i' t'i'->'o'; {:
   1.1.1             rule: i'ni' t'ni'->'ti';
   1.1.2             rule: i'ti' t''->''; :}
   1.2          rule: i'l' t'l'->'ti'; :}
```

```
3  1      :--->:RULE: if("") then(""-->"") exc(2); {:
                 :
   1.1           :--->:RULE: if("i") then("i"-->"o") exc(2); {:
                 :      :
   1.1.1         :      :--->:RULE: if("ni") then("ni"-->"ti");
   1.1.2         :      :--->:RULE: if("ti") then(""-->""); :}
                 :
   1.2           :--->:RULE: if("l") then("l"-->"ti"); :}
```

```
4  1      če je končnica ''                              //rule: i''   t''->''
              razen če je končnica tudi:                 //{:
   1.1            'i'                                    //rule: i'i'  t'i'->'o'
                      razen če je končnica tudi:         //{:
   1.1.1                  'ni' jo zamenjaj z 'ti'        //rule: i'ni' t'ni'->'ti'
   1.1.2                  'ti' jo zamenjaj z 'ti'        //rule: i'ti' t''->''
                      jo zamenjaj z 'o'                  //:}
   1.2            'l' jo zamenjaj z ''                   //rule: i'l'  t'l'->'ti'
              jo zamnjaj z ''                            //:}
```

```
5  1      `--> RULE:(ruleid("1") ending("") trans(""-->"") ); {:
   1.1      |--> RULE:(ruleid("1.1") ending("i") trans("i"-->"o") ); {:
   1.1.1    | |--> RULE:(ruleid("1.1.1") ending("ni") trans("ni"-->"ti") );
   1.1.2    | `--> RULE:(ruleid("1.1.2") ending("ti") trans(""-->"") ); :}
   1.2      `--> RULE:(ruleid("1.2") ending("l") trans("l"-->"ti") ); :}
```

3.3 STRUKTURA ALGORITMA TER IMPLEMENTACIJA

Kot rečeno smo program zasnovali po načelih standardne blok sheme prevajalnika (diagram 3.1). Vendar smo postopek zaradi specifičnosti problema nekoliko prilagodili. Ker naš jezik vsebuje zelo malo kontekstno neodvisnih elementov, ne potrebujemo ločene semantične analize, ki se ukvarja s tem. Ta je združena kar v sintaksnem analizatorju. Prav tako ne potrebujemo prevajanja v vmesno kodo pred optimizacijo, saj že sintaksna analiza izdela vmesno drevo, na katerem delamo optimizacijo. Prevajanje v končno, strojno kodo smo poimenovali serializacija, ker gre v bistvu za prevajanje optimiziranega drevesa v zaporedje bajtov.

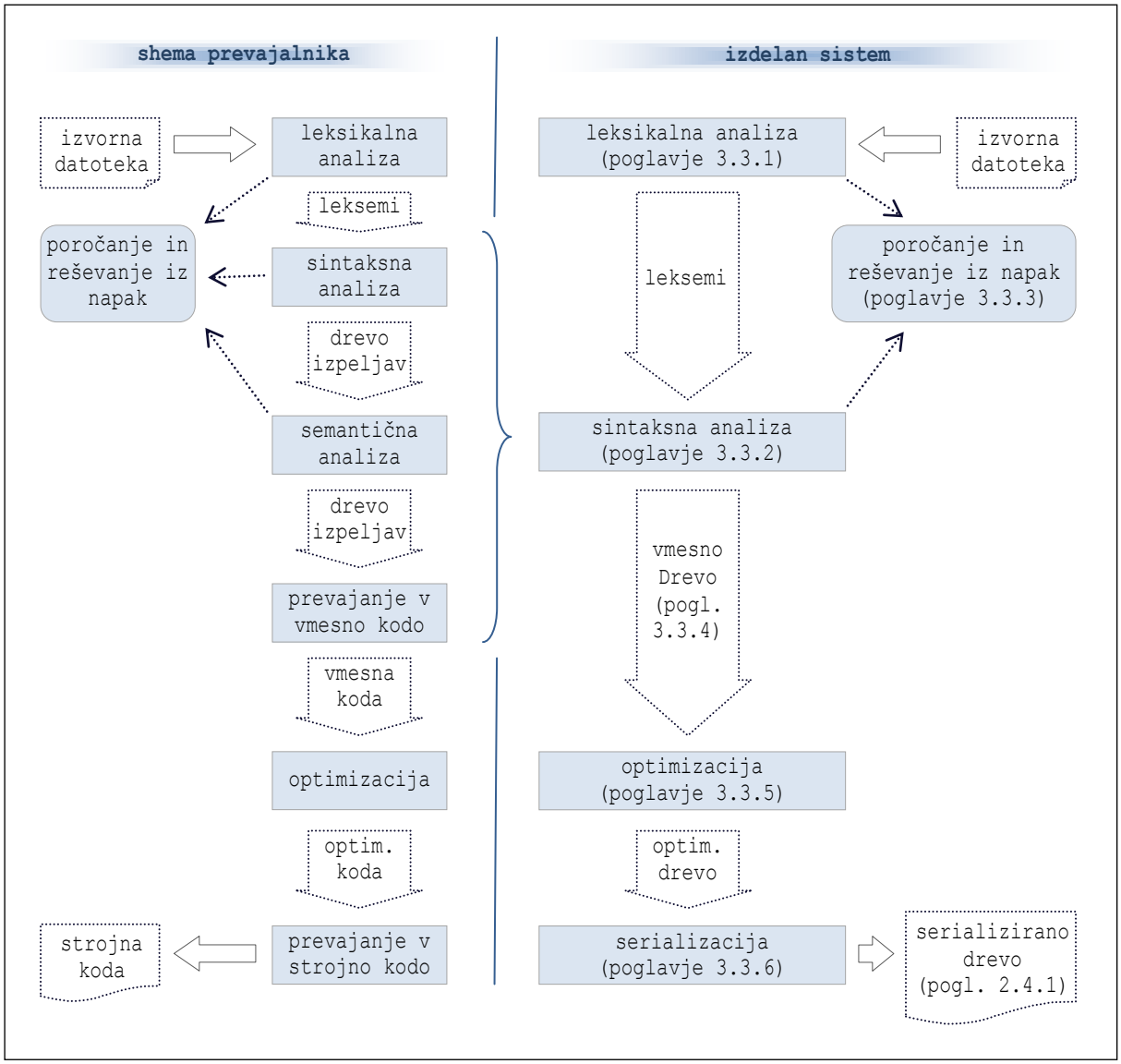


DIAGRAM 3.1: SPLOŠNA BLOK SHEMA PREVAJALNIKA V PRIMERJAVI Z NAŠIM SISTEMOM

3.3.1 LEKSIKALNA ANALIZA

Leksikalna analiza je prva stopnja prevajanja vhodne datoteke. Algoritem, ki jo izvaja, imenujemo tudi leksikalni analizator. Naloga tega analizatorja je vhodno zaporedje znakov, ki jih zaporedoma bere iz datoteke, logično združiti v večje pomenske enote npr. ključne besede, številke, operatorje, komentarje… Te enote imenujemo leksemi *(angl. lexeme)* in so osnovni gradniki sintaksne analize.

Leksikalni analizator smo izdelali z orodjem *Flex++ (fast lexical analyzer generator c/c++)*, ki je zasnovan na programu *Flex* in je spremenjen tako, da omogoča izdelavo kode za c++. Orodje omogoča avtomatsko pretvorbo definicije leksikalnega analizatorja v kodo razreda, ki ga implementira. Zato ni potrebno pisati kode ampak samo definicijo analizatorja. S tem izločimo veliko napak, ki so sicer neizogibne v tako dolgem programu.

Definicijsko datoteko za *Flex++* sestavlja več sekcij. Na tem mestu bomo pokazali dve, ki sta za definicijo leksikalne analize najpomembnejši. V kodi 3.1 najprej definiramo zaporedja znakov (lekseme), v kodi 3.2 pa pravila, s katerimi iščemo lekseme v datoteki. Definicije leksemov sicer niso obvezne, saj lahko v pravilih direktno uporabljamo zaporedja znakov, vendar je tak pristop elegantnejši.

KODA 3.1: LEKSIKALNI ANALIZATOR (LEKSEMI)

```
1  WhtSpcChar  [\f\b\t\v ]*        //FormFeed, Backspace, HorizontalTab, VerticalTab, Space
2  NewLine     \r|\n|\r\n          //CarriageReturn, NewLine, CarriageReturn & Newline
3  Comment     [^\r\nRr{:]*|[Rr{:] //vsi razen NL, CR, začetek od 'rule','{:',':}'
4
5  //Decimal, String and Char konstante
6  DecStart    1|2|3|4|5|6|7|8|9
7  DecDigit    0|{DecStart}
8  CharChar    [^\n\r\'\\]|(\\.)
9  StringChar  [^\n\r\"\\]|(\\.)
10
11 IntConst    {DecStart}({DecDigit}){0,8}
12 ChrConst    \'{CharChar}*\'
13 StrConst    \"{StringChar}*\"
14
15 //ključne besede (rule:)  (n, name, ruleid, id)  (i, if, end ending, suf, suffix)
16 //               (t, then) (e, exc, except, exceptions)
17 Rule        [Rr][Uu][Ll][Ee]:
18 Id          [Nn]([Aa][Mm][Ee])?|([Rr][Uu][Ll][Ee])?[Ii][Dd]
19 If          [Ii][Ff]?|[Ee][Nn][Dd]([Ii][Nn][Gg])?|[Ss][Uu][Ff]([Ff][Ii][Xx])?
20 Then        [Tt]([Hh][Ee][Nn])?|[Tt][Rr][Aa][Nn][Ss]([Ff][Oo][Rr][Mm])?
21 Exc         [Ee]([Xx][Cc]([Ee][Pp][Tt]([Ii][Oo][Nn][Ss])?)?)?
22
23 //seznam izjem začetek="{:", konec=":}"
24 ExcStart    [{][:]
25 ExcEnd      [:][}]
26
27 //operatorji
28 Semicol     ";"
29 Lbrac       "("
30 Rbrac       ")"
31 Transf      "-"+">" //(->, -->, --->, ---->, ...)
```

Koda 3.1 je v glavnem enostavno razumljiva, zato tukaj navajamo le najpomembnejše definicije:

- Ključne besede: (velikost znakov ni pomembna):
  - Začetek opisa pravila *{Rule}* ima le eno možno vrednost "rule:".
  - Lastnost identifikator pravila *{Id}*, možne vrednosti: "n" "name", "rule", "ruleid".
  - Lastnost končnica *{If}*, možne vrednosti: "i", "if", "end", "ending", "suf", "suffix".
  - Lastnost transformacija *{Then}*, možne vrednosti: "t", "then", "trans", "transform".
  - Lastnost število izjem *{Exc}*, možne vrednosti: "e", "exc", "except", "exceptions".
- Seznam izjem:
  - Začetek seznama *{ExcStart}*, možna vrednost je "{:".
  - Konec seznama *{ExcEnd}*, možna vrednost je ":}".

- Komentar: *{Comment}*
  - Zaporedje poljubnih znakov razen prvega znaka za *{Rule}* ("R" ali "r"), prvega znaka za *{ExcStart}* ali *{ExcEnd}* ("{" ali ":") ter preloma vrstice ("\r" ali "\n").
  - Eden izmed zgoraj izključenih znakov ("R", "r", "{", ":").

Generirani leksikalni analizator deluje tako, da poskuša najti tisto definicijo, ki pokrije najdaljšo vrednost. Zato je definicija komentarja malenkost kompliciran, saj ne sme pokrivati recimo začetka definicije pravila. Analiza tako prepozna vse znake do prvega "r", zaključi definicijo komentarja, skuša pokriti definicijo pravila "rule:" in če mu uspe, vrne pravilo, sicer pa komentar z vrednostjo "r". Vrednost "generated rule:" tako razbije na *{Comment}* "gene", *{Comment}* "r", *{Comment}* "ated ", *{Rule}* "rule:".

Koda 3.2 določa pravila, po katerih prepoznavamo tekst. Definirali smo dve stanji, ki sprejemata različne lekseme:

- *<COMMENT>* predstavlja stanje, v katerem sprejemamo poljubni tekst. V kolikor ga prepoznamo kot *{Rule}, {ExcStart}* ali *{ExcEnd},* ga vrnemo sintaksni analizi, sicer pa proglasimo za *{Comment}* ali *{NewLine}* ter ignoriramo. Če pridemo do konca datoteke *<<EOF>>,* zaključimo leksikalno analizo. V stanju *<COMMENT>* se nahajamo na začetku vsake vrstice in smo v tem stanju, dokler ne prepoznamo začetka definicije pravila *{Rule}* (vr. 3). Takrat se avtomat premakne v stanje *<RULE>*, v katerem se nahajamo, dokler ne zaključimo branja pravila. V stanju *<COMMENT>* tako ne moremo naleteti na napako ne glede na tekst ki ga sprejemamo. To omogoča močno razširitev oblike datoteke.
- *<RULE>* je stanje, v katerem se nahajamo, ko skušamo sprejeti pravilo in njegove lastnosti. V tem stanju imamo natanko predpisano, katere nize lahko sprejemamo (vr. 10-27). Neprepoznan niz se izraža kot napaka (vr. 29). Dovoljene so naslednje skupine:
  - Ključne besede za lastnosti (vr. 10-13).
  - Operatorji: oklepaji "(", ")" ter znak za transformacijo *{Transf}* (vr. 15-17).
  - Presledki (vr. 19).
  - Črkovne in numerične konstante (vr. 21-23).

  Iz stanja *<RULE>* se vrnemo v *<COMMENT>* na tri načine:
  - Ko vidimo znak za konec pravila ";". Tako lahko v isto vrstico navedemo več pravil.
  - Znak za prelom vrstice. Pravilo se zaradi tega ne more raztezati preko več vrstic.
  - Konec datoteke.

Spremenljivke, ki jih vračamo (rdeče v kodi 3.2), se nanašajo na lekseme, ki jih po klicu *return* obravnava sintaksni analizator in so v kodi 3.3 prav tako označeni z rdečo barvo.

KODA 3.2: LEKSIKALNI ANALIZATOR (PRAVILA)

```
1  //COMMENT je stanje leksikalnega analizatorja, ko analiziramo komentarje
2  <COMMENT>{Comment}  //ignoriraj
3  <COMMENT>{Rule}     BEGIN RULE; return RULE_START;
4  <COMMENT>{ExcStart} return EXC_START;
5  <COMMENT>{ExcEnd}   return EXC_END;
6  <COMMENT>{NewLine}  //ignoriraj
7  <COMMENT><<EOF>>    exit;
8
9  //RULE je stanje leksikalnega analizatorja, ko analiziramo pravilo
10 <RULE>{Id}          return ID;
11 <RULE>{If}          return IF;
12 <RULE>{Then}        return THEN;
13 <RULE>{Exc}         return EXC;
14
15 <RULE>{Lbrac}       return LBRAC;
16 <RULE>{Rbrac}       return RBRAC;
17 <RULE>{Transf}      return TRANSF;
18
19 <RULE>{WhtSpcChar}  //ignoriraj
20
21 <RULE>{StrConst}    return STRING;
22 <RULE>{ChrConst}    return STRING;
23 <RULE>{IntConst}    return INT;
24
25 <RULE>{Semicol}     BEGIN COMMENT; return RULE_END;
26 <RULE>{NewLine}     BEGIN COMMENT; return RULE_END;
27 <RULE><<EOF>>       BEGIN COMMENT; return RULE_END;
28
29 <RULE>.             return ERROR;
```

3.3.2 SINTAKSNA ANALIZA

Naloga sintaksne analize je združevati lekseme v večje pomenske enote (stavke) in tako zgraditi drevo izpeljav za celotno datoteko. Sintaksni analizator dobiva lekseme neposredno od leksikalnega, izhod pa je v našem primeru kar vmesno drevo.

Ker sta komponenti leksikalnega in sintaksnega analizatorja tesno povezani, je v primeru avtomatsko izdelanih komponent potrebna njuna kompatibilnost ter zmožnost sodelovanja. Zaradi tega smo uporabili tudi generator sintaksnih analizatorjev *Bison++* izpeljan iz GNU projekta *Bison*. Koncept delovanja je podoben kot pri *Flex*-u. Iz definicijske datoteke sintaksnega analizatorja *Bison* generira implementacijo razreda v c++, ki zna v sodelovanju s leksikalnim analizatorjem (dobljenim s pomočjo *Flex++*) sintaksno analizirati datoteke. Kot zgled koristnosti takega orodja lahko navedemo, da v našem primeru *Bison++* generira skoraj 2000 vrstic izvorne kode iz samo 200 vrstic dolge ter pregledne definicijske datoteke.

Koda 3.3 prikazuje najpomembnejši del definicije sintaksnega analizatorja, to so pravila, po katerih se generira drevo izpeljav. Pravila so približno urejena od najbolj specifičnega do najsplošnejšega.

KODA 3.3: SINTAKSNI ANALIZATOR

```
1  //vrednosti lastnosti (črkovne in numerične konstante)
2  string  : STRING | LBRAC STRING RBRAC;
3  int     : INT | LBRAC INT RBRAC;
4
5  //lastnosti
6  id      : ID string;
7  if      : IF string;
8  then    : THEN thenrule | THEN LBRAC thenrule RBRAC;
9  thenrule: STRING TRANSF STRING;
10 exc     : EXC int;
11
12 //seznam lastnosti
13 proplist: /*prazen*/ | proplist id | proplist if | proplist then | proplist exc;
14 propert : proplist | LBRAC proplist RBRAC;
15
16 //pravila
17 rule    : RULE_START propert RULE_END;
18 rdr     : rule except;
19
20 //izjeme
21 except  : /*prazen*/ | EXC_START rdrlist EXC_END;
22 rdrlist : /*prazen*/ | rdrlist rdr;
23
24 //celotna datoteka
25 file    : rdr;
```

Za lažje razumevanje, kako poteka sintaksna analiza po pravilih iz kode 3.3, sledijo opisi nekaj pomembnejših stavčnih oblik:

- Začetek sintaksne analize datoteke se vedno začne pri stavčni obliki *[file]* (vr. 25).
- Oblika *[file]* se lahko razvije le v *[rdr]* ta pa v stik *[rule]* in *[except]* (vr. 18). Iz tega sledi, da mora imeti vsaka datoteka na prvem nivoju natančno eno (korensko) pravilo.
- Oblika *[except]* (vr. 21) predstavlja seznam izjem. Ker se lahko seznam razvije v prazno stavčno obliko, to pomeni, da ni nujno, da ima vsako pravilo seznam izjem. Po drugi strani pa lahko seznam izjem za pravilo obstaja *[rdrlist]* (vr. 22), vendar je ta prazen. V to obliko bi se recimo razvil niz "{: :}". Seveda se *[rdrlist]* lahko razvije v poljubno dolg niz oblik *[rdr]*.
- *[rule]* je oblika, ki predstavlja natančno eno pravilo (vr. 17). Začne se z "rule:", konča pa z ";", novo vrstico ali koncem datoteke (kodi 3.2 vr. 25-27). V svojem jedru pa vsebuje lastnosti *[propert]*.
- *[propert]* se razvije v seznam izjem *[proplist]*, ki so lahko zaprte v oklepaj ali ne (vr. 14). Tako sprejemamo vrednosti "rule: suf('a') trans('a'->'o');" in tudi "rule:(suf('a') trans('a'->'o'));". *[proplist]* predstavlja tudi edino kontekstno odvisno pravilo (vr. 13). Preverjamo namreč, če sta obvezni lastnosti [if] in [then] prisotni ter če je kakšna lastnost prisotna dvakrat.
- Vrstice od 6 do 10 opisujejo stavčne oblike za lastnosti *[id]*, *[if]*, *[then]* in *[exc]*. Lastnosti so sestavljene iz ključne besede in vrednosti. Vrednost vseh lastnosti je lahko ali pa ni zaprta

v oklepaje. Tako sta lastnosti "exc(8)", "then('a'->'o')" enakovredni "exc 8", "then 'a'->'o'" in tudi "exc8", "then'a'->'o'".

- Vrednosti lastnosti so niz znakov *[string]* (vr. 2) za *[id]* in *[if]*, celoštevilska konstanta *[int]* (vr. 3) za *[exc]* in sestavljena *[thenrule]* (vr. 9) za *[then]*.

Primer 3.4 prikazuje začetek sintaksne analize neke datoteke pravil. Lahko vidimo, da se drevo izpeljav precej napihne. Vendar je v našo sintaksno analizo implicitno vgrajena tudi stopnja prevajanja v vmesno kodo (diagram 3.1). Tako se celotna drevesna hierarhična struktura stavčne oblike *[rule]* stisne v en objekt, ki se v izvorni kodi imenuje *RdrRule*.

3.3.3 Reševanje iz napak in poročanje

PRIMER 3.2: DREVO Z NAPAKAMI

```
1  RULE:( suffix("") transform(""-->"") except(1) ) comment; {:
2   `---> RULE:( id("pr2") suffix("i") transform("i"-->"o") except(2) id"pr3" ); {:
3          |---> RULE:( suffix transform("ni"-->"ti") );
4          |---> RULE:( sufix("ni") transform("ni"-->"ti") );  comment
5          `---> RULE:( suffix("ti") ); :}
6           :}
```

PRIMER 3.3: POROČILO O NAPAKAH ALGORITMA ZA GRADNJO LEMATIZATORJEV

```
1   Parse unsuccessful!
2     Encountered 6 error(s).
3     Output of first 6 error(s):
4     [#1] parse error! (line:1, col:56, symb:"c")
5         `---> RULE:( suffix("") transform(""-->"") except(1) ) comment; {:
6                                                               ^
7     [#2] property 'name' was already set! (line:2, col:82, symb:")")
8         ...nsform("i"-->"o") except(2) id"pr3" ); {:
9                                             ^
10    [#3] parse error! (line:3, col:35, symb:"transform")
11                      |---> RULE:( suffix transform("ni"-->"ti") );
12                                    ^^^^^^^^^
13    [#4] parse error! (line:4, col:31, symb:"i")
14                      |---> RULE:( sufix("ni") transform("ni"-->"ti") );  comm...
15                                  ^
16    [#5] property 'then' is mandatory! (line:5, col:44, symb:":}")
17                      `---> RULE:( suffix("ti") ); :}
18                                                   ^^
19    [#6] parse error! (line:8, col:2, symb:"")
20
21          ^
```

Za vsak prevajalnik je zaželeno, da ima čim boljšo podporo za reševanje iz napak. Ko leksikalni ali sintaksni analizator naleti na napako v vhodni datoteki, se poskuša napačni del preskočiti in nadaljevati analizo na naslednjem delu, ki ga prepoznamo kot pravilnega. V našem primeru je osnovna enota pravilo, zato v primeru napake preskočimo trenutno pravilo in skušamo najti naslednje. Nepravilnosti se lahko pojavijo tudi pri definiciji seznama izjem. Primer 3.2 prikazuje napačno drevo, ki je vhod, 3.3 pa odziv algoritma na to drevo. Koliko napak program najde in koliko jih izpiše, se nastavlja preko stikal, privzeta vrednost pa je 100 najdenih in 7 izpisanih napak.

V večini primerov informacija iz poročila napak zadošča, da napako odpravimo. Seveda pa lahko naredimo tudi take napake, ki sistem za reševanje zmedejo in zato poroča neke nepovezane napake. Podobno velja tudi za druge, mnogo bolj premišljeno zasnovane prevajalnike kot je npr. prevajalnik za c++.

3.3.4 VMESNO DREVO

V vmesnem drevesu, ki ga generira sintaksna analiza, so le hierarhični odnosi, ki jih implicira stavek *[rdr = rule · except]* (koda 3.3 vr. 18).

PRIMER 3.4: ZAČETEK SINTAKSNE ANALIZE DATOTEKE PRAVIL

```
1                   vr. 25  file = "RULE:( if('a') then('a'-->'o') except 21 ); {: ... :}"
1.1                 vr. 18    rdr = "RULE:( if('a') then('a'-->'o') except 21 ); {: ... :}"
1.1.1               vr. 17      rule = "RULE:( if('a') then('a'-->'o') except 21 );"
1.1.1.1                           RULE_START = "RULE:"

1.1.1.2             vr. 14        propert = "( if('a') then('a'-->'o') except 21 )"
1.1.1.2.1                           LBRAC = "("

1.1.1.2.2           vr. 13          proplist = "if('a') then('a'-->'o') except 21"
1.1.1.2.2.1         vr. 13           proplist = "if('a') then('a'-->'o')"
1.1.1.2.2.1.1       vr. 13              proplist = "if('a')"
1.1.1.2.2.1.1.1     vr. 13                proplist = ""
1.1.1.2.2.1.1.1.1                           /*prazen*/ = ""

1.1.1.2.2.1.1.2      vr. 7                if = "if('a')"
1.1.1.2.2.1.1.2.1                           IF = "if"
1.1.1.2.2.1.1.2.2    vr. 2                string = "('a')"
1.1.1.2.2.1.1.2.2.1                         LBRAC = "("
1.1.1.2.2.1.1.2.2.2                         STRING = "'a'"
1.1.1.2.2.1.1.2.2.3                         RBRAC = ")"

1.1.1.2.2.1.2        vr. 8              then = "then('a'-->'o')"
1.1.1.2.2.1.2.1                           THEN = "then"
1.1.1.2.2.1.2.2                           LBRAC = "("
1.1.1.2.2.1.2.3      vr. 9                thenrule = "'a'-->'o'"
1.1.1.2.2.1.2.3.1                           STRING = "'a'"
1.1.1.2.2.1.2.3.2                           TRANSF = "-->"
1.1.1.2.2.1.2.3.3                           STRING = "'o'"
1.1.1.2.2.1.2.4                           RBRAC = ")"

1.1.1.2.2.2         vr. 10             exc = "except 21"
1.1.1.2.2.2.1                           EXC = "except"
1.1.1.2.2.2.2        vr. 3              int = "21"
1.1.1.2.2.2.2.1                           INT = "21"

1.1.1.2.3                           RBRAC = ")"
1.1.1.3                           RULE_END = ";"

1.1.2               vr. 21      except = "{: ... :}"
1.1.2.1                           EXC_START = "{:"
1.1.2.2             vr. 22        rdrlist = "..."
1.1.2.2.(...)                       ...
1.1.2.3                           EXC_END = ":}"
```

Vmesno drevo si lahko predstavljamo kot splošno RDR drevo, npr. iz primera 1.4. Vsako vozlišče je predstavljeno z objektom *RdrTree*, ki ima poleg drugih lastnosti (objekt *RdrRule*) tudi seznam vseh podrejenih vozlišč (izjem pravila).

V dejanski implementaciji ima razred *RdrTree* poleg podatkov tudi vse potrebne funkcije za optimizacijo in serializacijo drevesa, pa tudi za testno lematizacijo, izpisovanje, ... Vendar pa podroben opis razreda presega okvir tega dela, zato bomo v naslednjih poglavjih predstavili le koncepte optimizacije in serializacije. Bralci, ki bi radi o tem izvedeli kaj več, najdejo kodo v elektronskih dodatkih diplomskega dela (priloga B).

3.3.5 Optimizacija

Težave, ki jih vsebuje popolnoma splošno RDR drevo, smo že omenili (2.2.2 Varianta 2. Popravljeno RDR drevo). V tem poglavju predstavimo njihovo podrobnejšo razčlenitev in koncept algoritma, ki jih odpravi.

Najpomembnejša zahteva te stopnje izdelave lematizatorja je zagotovljena ekvivalenca izhodiščnega in optimiziranega drevesa. Da je semantika popravljenega drevesa natančno enaka semantiki originalnega, je namreč pogoj izdelave lematizatorja, torej celotnega sistema, ki ga opisujemo v tem poglavju.

Sklici na pravila v nadaljevanju tega podpoglavja se, če ni navedeno drugače, nanašajo na primer 3.5. Za prikaz algoritma optimizacije smo drevo iz primera 1.4 dopolnili še z dvema, neželenima lastnostnima (pr. 1.4.1, 1.6, 1.7).

Primer 3.5: RDR drevo pred optimizacijo

```
1       1  RULE:( suffix("") transform(""-->"") except(5) ); {:
1.1     2   |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
1.2     3   |---> RULE:( suffix("ši") transform("ši"-->"sati") );
1.3     4   |---> RULE:( suffix("šimo") transform("šimo"-->"sati") );
1.4     5   |---> RULE:( suffix("l") transform("l"-->"ti") ); {:
1.4.1   6   |       `---> RULE:( suffix("u") transform("u"-->"o") ); :}
1.5     7   |---> RULE:( suffix("i") transform("i"-->"o") except(2) ); {:
1.5.1   8   |       |---> RULE:( suffix("ni") transform("ni"-->"ti") );
1.5.2   9   |       `---> RULE:( suffix("ti") transform(""-->"") ); :}
1.6    10   |---> RULE:( suffix("ni") transform("ni"-->"ti") );
1.7    11   '---> RULE:( suffix("i") transform("i"-->"a") ); :}
```

Lastnosti splošnega RDR drevesa, ki jih želimo optimizirati, so naslednje:

- Velika stopnja vejitve v začetnih vozliščih. Če drevo generiramo z algoritmom [12], imajo začetna vozlišča izredno veliko izjem (preko 500 pri podatkih za slovenščino). Vendar lahko v vsakem vozlišču izjeme ločimo glede na dodano črko (DČ, definirano v 2.3.1 Notacija). Z uporabo tega načina ločevanja bi bila stopnja vejitve maksimalno 25 oz. toliko, kolikor je vseh črk abecede.

- Zaporedno iskanje izjem. Po definiciji RDR koncepta je potrebno izjeme pravila iskati od začetka proti koncu. V želji, da bi pohitrili postopek, potrebujemo direktni način iskanja izjem. Tudi tukaj vidimo, da rešitev nudi ločevanje izjem glede na DČ. Med lematizacijo lahko vsaki besedi v vsakem vozlišču enolično določimo DČ in z njeno pomočjo najdemo morebitno izjemo pravila.
- Nedostopna vozlišča. V določenih primerih se lahko v drevesu pojavijo vozlišča, do katerih algoritem ne more dostopati. Problem se izraža v dveh različicah:
  - Zaradi zaporednosti iskanja izjem imajo večjo prioriteto začetna vozlišča. Zato logično podrejena pravila (pr. 1.6) in logično ekvivalentna pravila (pr. 1.7), ki so navedena v istem seznamu izjem kot prvo tako pravilo (pr. 1.5), nikoli ne pridejo do evalvacije. Takšna lahko brez posledic odstranimo.
  - Zaradi hierarhije se med izjemami pravila lahko pojavijo tudi takšna pravila, ki sploh nimajo enakega korena (pr. 1.4.1). Tudi takšna lahko eliminiramo.

Prvi dve točki opisujeta sistemski težavi RDR koncepta, ki se pojavita tudi pri avtomatsko generiranih drevesih. Problem iz zadnje točke se v takih drevesih ne bi smel pojaviti, saj bi moral dobro zasnovani učni algoritem že sam izločiti takšne primere, oz. jih sploh ne bi smel generirati. Zlahka pa se pojavijo kot napake pri ročno izdelanem drevesu. Ker ne gre za sintaktične nepravilnosti datoteke, algoritem ne javi napake temveč samo popravi drevo z izločanjem takih primerov.

Oris algoritma optimizacije:

- V nespremenjenem drevesu oštevilčimo vsa vozlišča na način "najprej v globino". Zgled takega oštevilčenja je podan v drugem stolpcu primera 3.5.
- Uredimo sezname izjem vseh vozlišč glede na končnico (suffix). Končnico urejamo leksikografsko vendar po črkah od zadaj naprej. Tako npr. 'za' leži pred 'ab'. Urejeno drevo prikazuje primer 3.6.

PRIMER 3.6: PRVI KORAK OPTIMIZACIJE

```
 1  RULE:( suffix("") transform(""-->"") except(5) ); {:
11   |---> RULE:( suffix("i") transform("i"-->"a") );
 7   |---> RULE:( suffix("i") transform("i"-->"o") except(2) ); {:
 8   |       |---> RULE:( suffix("ni") transform("ni"-->"ti") );
 9   |       `---> RULE:( suffix("ti") transform(""-->"") ); :}
10   |---> RULE:( suffix("ni") transform("ni"-->"ti") );
 3   |---> RULE:( suffix("ši") transform("ši"-->"sati") );
 5   |---> RULE:( suffix("l") transform("l"-->"ti") ); {:
 6   |       `---> RULE:( suffix("u") transform("u"-->"o") ); :}
 2   |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
 4   `---> RULE:( suffix("šimo") transform("šimo"-->"sati") ); :}
```

- V tem koraku primerjamo končnice vozlišč. Vedno primerjamo le sosednji vozlišči, saj ureditev zagotavlja, da logično podrejena, nadrejena in ekvivalentna vozlišča ležijo skupaj. Na prvem nivoju primera 3.6 bi tako primerjali pare <11, 7>, <7, 10>, <10, 3>, <3, 5>, <5,

2> ter <2, 4>. S tem si znižamo kompleksnost problema iz kvadratne zahtevnosti v linearno. Primerjave, ki jih delamo, so naslednje:

- Če je končnica obeh pravil enaka, potem zbrišemo pravilo z večjo zaporedno številko[2]. V paru <11, 7> tako zbrišemo 11.
- Če je končnica enega primera logično podrejena drugemu, potem:
  - če ima podrejeni primer višjo zaporedno številko kot nadrejeni, zbrišemo podrejenega[3]. V paru <7, 10> je 10["ni"] podrejen do 7["i"], zato ga zbrišemo.
  - če ima podrejeni primer nižjo zaporedno številko kot nadrejeni, dodamo logično podrejeni primer v seznam izjem nadrejenega[4]. Ker smo zbrisali 10, zdaj sosednji par postane tudi <7,3>, vendar ima tukaj podrejeni 3["ši"] nižjo številko od nadrejenega 7["i"], zato ga dodamo med izjeme.
- Kadar končnici nista enaki niti podrejeni, imata pa enak zadnji del (končnico končnice), naredimo novo vozlišče, katerega končnica je ta skupni del, transformacija pa enaka transformaciji nadrejenega vozlišča[5]. Nastavimo zaporedno številko na maksimalno možno vrednost, tako da bomo temu vozlišču dodali vsa morebitna nadaljnja vozlišča z enakim skupnim delom. Paru <2,4> dodamo novo nadrejeno vozlišče 12 (primer 3.7).

- Zadnja dva koraka, urejanje vozlišč in primerjanje končnic, ponavljamo rekurzivno na podrejenih vozliščih. Rezultat je prikazan na primeru 3.7.

PRIMER 3.7: DRUGI KORAK OPTIMIZACIJE

```
 1  RULE:( suffix("") transform(""-->"") except(5) ); {:
 7   |---> RULE:( suffix("i") transform("i"-->"o") except(2) ); {:
 8   |       |---> RULE:( suffix("ni") transform("ni"-->"ti") );
 3  .|       |---> RULE:( suffix("ši") transform("ši"-->"sati") );
 9   |       `---> RULE:( suffix("ti") transform(""-->"") ); :}
 5   |---> RULE:( suffix("l") transform("l"-->"ti") ); {:
 6   |       `---> RULE:( suffix("u") transform("u"-->"o") ); :}
12   `---> RULE:( suffix("mo") transform(""-->"") ); {:
 2          |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
 4          `---> RULE:( suffix("šimo") transform("šimo"-->"sati") ); :}
             :}
```

[2] Pravilo z večjo zaporedno številko je namreč v originalnem drevesu ležalo za tistim z nižjo, zato algoritem nikoli ne bi mogel dostopati do njega.

[3] Podobno kot v opombi 2. Takšno pravilo je nedostopno.

[4] Podrejeni je bil v originalnem drevesu naveden pred nadrejenim. V primeru lematizacije besede, ki ustreza podrejenemu, se je iskanje pravila usmerilo v podrejeni primer, še preden je algoritem prišel do nadrejenega. V novem drevesu bo iskanje najprej upoštevalo nadrejeno pravilo, a ker to zdaj vsebuje podrejeno med izjemami, se bo iskanje nadaljevalo v tej izjemi.

[5] Tudi to dodano vozlišče zagotavlja nespremenjeno semantiko drevesa. Če beseda ustreza končnici enega izmed teh dveh pravil, potem bodo izjeme novega pravila poskrbele, da bo ta beseda našla pot do pravega pravila. Če pa bo končnica ustrezala le novemu pravilu in nobeni izjemi, potem se bo izvedla transformacija starša, enako kot bi se sicer.

- Sledi odstranjevanje zaradi hierarhije nedostopnih vozlišč. Z rekurzivnim obiskom iz vozlišč odstranimo vse tiste izjeme, ki niso logično podrejene njihovim staršem. V primeru 3.7 tako zbrišemo vozlišče 6.
- Zadnji korak je brisanje nepotrebnih vozlišč. Nepotrebna so vsa tista vozlišča, ki imajo nič ali eno izjemo, transformacija takega vozlišča pa mora biti enaka nadrejenemu vozlišču. To vozlišče lahko zbrišemo, morebitno izjemo pa dodamo izjemam nadrejenega vozlišča.

Končni rezultat optimizacije tega zgleda je prikazan v primeru 2.1. Ker smo zagotovili, da vsak korak posebej ohranja semantično identičnost drevesa, lahko to trdimo tudi za celotni postopek. Poleg tega pa smo tezo preizkusili tudi eksperimentalno, tako da smo na veliko naključno generiranih drevesih lematizirali naključno generirane besede z originalnim in optimiziranim drevesom. Poleg naključnih smo uporabili tudi podatke za vse jezike, ki smo jih imeli na voljo. Tudi eksperimentalno se je teza potrdila kot pravilna.

3.3.6 SERIALIZACIJA

Natančno obliko serializiranih podatkov podaja poglavje 2.4.1. Ker je bil način kodiranja tako vozlišč kot tudi celotnega drevesa že opisan, na tem mestu ostane le še oris postopka določitve naslovov vozlišč v tabeli.

Problem naslovov je namreč v tem, da se vozlišča vedno sklicujejo po tabeli naprej, zato je potrebno naslove vedeti še preden polnimo vozlišča v tabelo:

- Najprej nastavimo dolžino naslovov na 3B.
- Obiskujemo vozlišča na način "najprej v globino" in izpuščamo tista vozlišča, ki ne bodo serializirana.
  - Ker imamo tipe vozlišč in dolžine podatkov v njih podrobno definirane, lahko za vsako vozlišče izračunamo njegovo dolžino v bajtih.
  - Naslov prvega vozlišča je 0.
  - Naslov vsakega ne-prvega vozlišča je naslov plus dolžina predhodno obiskanega.
  - Naslove shranimo v pripadajoče objekte (*RdrTree*).
- Ko izračunamo naslove vseh vozlišč, po enakem postopku izračunamo še naslov vseh pravil (vozlišč tipa 0) iz seznama pravil. Prvo pravilo dobi tisti naslov v tabeli, kjer se končajo podatki drevesa.
- Če se med postopkom slučajno zgodi, da je naslov večji od razpoložljivih 3B, potem povečamo dolžino naslova na 4B in ponovno poženemo algoritem od druge točke naprej.

S tako izračunanimi naslovi dobimo tudi dolžino celotne strukture. Preostane nam le še, da inicializiramo tabelo bajtov in vanjo po poljubnem postopku zapišemo vsa vozlišča in pravila, katera imajo dodeljen naslov.

3.4 GENERIRANJE KODE

Program torej izdela strukturo (serializirano drevo) za lematizator opisan v poglavju . Načinov za izvoz tega drevesa pa je več. Odločili smo se za naslednje tri izhodne predstavitve lematizatorja:

- Binarna datoteka: uporabimo jo, kadar imamo lematizator že preveden in uporabljen samostojno ali v večjem sistemu. S tem načinom v obstoječi lematizator enostavno uvozimo podatke za novo RDR drevo. Ponovno prevajanje ni potrebno. Uvoz podatkov je možen preko funkcije *LoadBinary* (koda 2.3, vr. 17).
- Zaglavna *(angl. header)* datoteka: če je koda lematizatorja del nekega večjega sistema, in če ne želimo, da bi moral uporabnik skrbeti za dodatno binarno datoteko, potem podatke vključimo kar v program. To dosežemo tako, da zaglavno datoteko pripnemo našemu sistemu in ponovno prevedemo program. Lematizator dobi na ta način nove podatke brez popravljanja njegove kode. Podroben specifikacijo uporabe tega načina podaja priloga A.
- Datoteka izvorne kode samostojnega lematizatorja: kadar želimo izdelati popolnoma samostojen program, ki bo takoj sposoben lematizacije besedil, uporabimo ta pristop. Generirana je izvršna koda, ki jo je potrebno prevesti. Lematizator bo privzeto uporabljal definicijo drevesa, ki smo jo naredili v tem načinu, možen pa bo tudi uvoz novih podatkov v obliki binarnih datotek. Uporaba takega lematizatorja je prav tako opisana v prilogi A.

Sistem generira datoteke glede na zgoraj naštete načine:

- Binarna datoteke: generiranje tega izhoda je najenostavnejše. Vse, kar je potrebno narediti, je direktni prepis tabele bajtov v datoteko. To se lahko stori s funkcijo *SaveBinary* (koda 2.3, vr. 16).
- Zaglavna datoteka: sistem jo generira tako, da šestnajstiškemu zapisu (predstavljen v kodi 2.2) pripne določene definicije predprocesorja c++ in vse skupaj zapiše v tekstovno datoteko. Če imamo objekt lematizatorja izdelan, lahko celotno kodo za zapis v zaglavno datoteko dobimo s funkcijo *ToStringHex* (koda 2.3, vr. 15). Ko tako zaglavno datoteko vključimo *(angl. include)* v datoteko razreda *RdrLemmatizer,* se podatki o novem serializiranem drevesu avtomatsko uporabijo kot privzeti podatki lematizatorja.
- Izvorna koda: izdelava se začne podobno kot v prejšnji točki. Generiramo zaglavno datoteko, nato pa jo združimo skupaj z datoteko izvorne kode razreda *RdrLemmatizator* in njegovo zaglavno datoteko. Ti dve datoteki delujeta v tem primeru kot nekakšno okostje novega lematizatorja, generatorju pa ju podamo kot parametra v komandni vrstici. Poleg združitve treh datotek pa sistem omogoči tudi *main* funkcijo, ki je potrebna za samostojno delovanje lematizatorja in je v splošnem v kodi onemogočena. To končno izvorno datoteko lahko prevedemo s standardnim c++ prevajalnikom in dobimo delujoč lematizator, katerega privzeto drevo je tisto, ki smo ga generirali v tem izvajanju. Tako izdelan lematizator pa seveda omogoča tudi uvoz novih lematizatorjev prek binarnih datotek.

4 UČENJE RDR PRAVIL

V tem poglavju predstavljamo zadnji izmed treh sklopov tega dela, ki so razvidni iz diagrama 1.1. Razvili smo nov način gradnje RDR dreves, za zgled in primerjalni algoritem pa smo uporabili algoritem, ki so ga razvili Joël Plisson in sodelavci [12]. Ta nam je bil v veliko pomoč za preverjanje pravilnosti in je na začetku služil kot neposredni vzorec za naš algoritem, a se je med razvojem pokazalo, da obstala elegantnejša metoda učenja. Novo razviti algoritem tako med učenjem RDR koncepta ne uporablja več. Edina povezava, ki je ostala nespremenjena, je oblika izhoda obeh algoritmov, to je RDR drevo. Implementacijo tega poglavja predstavlja modul *LemLearn*, njegov opis pa najdete v prilogi A.

Prva dva sklopa diplomske naloge sta med seboj zelo tesno povezana. Tako recimo, ne bi mogli izdelati lematizatorja brez generatorja in tudi generator ne bi imel pomena, če lematizator ne bi obstajal. To poglavje pa je kljub, še vedno tesni vsebinski prepletenosti, bolj samostojno in rezultate se lahko uporablja tudi brez povezave z ostalima dvema. Modul, ki povezuje vse tri sklope se imenuje *LemXval* (priloga A), opravlja pa funkcijo prečnega preverjanja točnosti naučenih dreves.

V nadaljevanju najprej predstavimo delovanje originalnega algoritma ter njegove lastnosti, nato pa pojasnimo kaj so bili razlogi za izdelavo novega. Koncept nato predstavimo podrobneje, podamo psevdokodo algoritma ter pokažemo delovanje na dveh primerih.

4.1 PODATKI IN IZHOD

Ker gre za učni algoritem so vhodni podatki množice primerov (pari morfoloških oblik ter njihovih lem). Oblika vhodne datoteke je povzeta po standardu iz baze Multext-East [5]. Ta oblika je podrobno opredeljena v podpoglavju 5.1.1 "Opis podatkov". Primer 5.1 podaja podmnožico parov, katere bomo v nadaljevanju uporabljali. Med razlago uporabimo zgolj pare obarvane modro v končnem primeru pa za nazornejši prikaz uporabimo kar vse pare.

Privzeti izhod je datoteka pravil skladna z definicijo vhodne datoteke iz poglavja 3.2 "Določitev vhodne datoteke". Izbiramo lahko med nekaj formati (trije so prikazani v primeru 3.1). To datoteko

lahko nato ročno pregledujemo, interpretiramo in popravljamo. Lahko pa iz nje z generatorjem izdelamo specifikacijo ali kodo lematizatorja.

4.2 OSNOVNI ALGORITEM

Osnovni algoritem je bil sprva implementiran v [12] ter reimplementiran v c++. Tako smo imeli odlično ter uniformno platformo za preizkušanje delovanja začetnega in našega novega algoritma.

Neposredno iz [12] prepisano psevdokodo algoritma podajamo v kodi 4.1. Komentar iz članka h kodi je še, da primere predstavljamo učnemu algoritmu enega za drugim.

KODA 4.1: PSEVDOKODA UČENEGA ALGORITMA [12]

```
1  if rule fires then
2      if correct_conclusion then
3          Do not need to add an exception.
4      else if incorect_conclusion then
5          Find differences with the example that induced the rule.
6          Create an exception to the rule that fired
7  else
8      Create a new subrule with an "else if" branch.
```

To idejo pa je možno poenostaviti. Privzamemo, da si že v začetku postavimo neko privzeto korensko pravilo, ki je tako splošno, da se proži na vsak primer. Stavek *else* iz vrstice 7 se potem nikoli ne proži, saj se vedno proži prej korensko pravilo. Stavki *else* so tako na prvem nivoju nepotrebni.

Jedro implementacije se naslanja na bistvo RDR koncepta, tj. funkcijo za iskanje proženega pravila. To funkcijo (*FindFiredRule*) smo predstavili v kodi 1.1 in omogoča elegantno učenje iz primerov. Psevdokoda našega implementiranega učnega algoritma je zapisana v kodi 4.2, pojasnilo pa je naslednje:

- 1: Začnemo s praznim najsplošnejšim korenskim pravilom.
- 2: Za vsak primer iz učne množice naredi:
  - 3: Najdi pravilo, ki se proži za dani primer.
  - 4,5: V kolikor pravilo napačno klasificira trenutni primer, dodaj novo izjemo temu pravilu. Končnica novega pravila se določi tako, da je le to bolj specifično od trenutnega pravila, hkrati pa je končnica daljša ali enaka končnici, ki jo potrebujemo za transformacijo iz besede v lemo.

KODA 4.2: PSEVDOKODA UČENJA RDR DREVESA

```
1  rootRule = new empty rule
2  foreach (example in learningSet)
3      firedRule = FindFiredRule(rootRule , example)
4      if (firedFule.Transform(example.word) != example.lemma)
5          firedFule.AddException(example.word, example.lemma)
```

Lematizacija poteka na popolnoma enak način, le da ne dodajamo novih izjem. Ta preprostost je tudi največja prednost te metode učenja.

Ko se lotimo implementacije, vidimo, da vseeno ni tako preprosta. Težava je, da dodajanje nove izjeme lahko pokvari predhodno pravilno lematizirane primere nekega vozlišča. Taki primeri lahko po naključju sovpadajo s pogojem nove izjeme tako, da bi v ponovnem poskusu iskanja proženega pravila našli novo izjemo, kar je napačno. V vsakem vozlišču moramo zato hraniti vse primere, ki so prožili to vozlišče. Ob dodajanju izjeme moramo še enkrat konsolidirati vse primere upoštevajoč novo izjemo. Ta postopek se izkaže za ozko grlo v smislu učinkovitosti učnega algoritma. Lahko si sicer pomagamo z raznimi naprednimi metodami shranjevanja in iskanja primerov znotraj vozlišč, recimo zgoščevalno tabelo, a to do precejšne mere zakomplicira implementacijo, težave pa še vedno ne odpravimo popolnoma.

4.3 Predlagane izboljšave

Med analizo delovanja smo ugotovili, da zgornji algoritem deluje v določenih situacijah precej naključno. Poleg tega pa smo želeli v algoritmu upoštevati tudi informacije, za katere smo sprevideli, da bi utegnile prispevati k točnosti izdelanega drevesa.

- Pri dodajanju novih izjem ter novih primerov v obstoječe izjeme algoritem ne upošteva pokritosti. Tako se lahko zgodi, da npr. v korenskem pravilu izdela izjemo za končnico 'a', katere transformacija pa je zelo netipična za ostale primere. Zaradi tega kasneje ta izjema pokrije zelo malo pravilnih primerov, kar povzroči, da ima tudi sama veliko izjem. Ker smo torej kot prvi primer videli besedo s končnico 'a', katere transformacija ni tipična, se je drevo zelo povečalo. Naša želja je, da algoritem pogleda, katera transformacija je najbolj značilna za 'a' in v to prvo izjemo vpiše to najbolj tipično pravilo.
- Naslednja težava se pojavi v vozliščih, kjer se primeri več ne ločijo med sabo. Končnice besed so kar enake besedam. Če so med temi primeri taki, ki imajo različno transformacijo, potem se pojavi problem, katero izmed teh transformacij privzeti kot veljavno v tem vozlišču. Osnovni algoritem se odloči kar za prvo transformacijo, na katero naleti. Podobno kot v prejšnji točki bi tukaj želeli dati prednost transformacijam, ki so bolj pogoste.
- Zmotilo nas je tudi neupoštevanje informacije o tem, da je končnica pravila, ki se proži, kar enaka celi besedi primera (beseda 'oba' v pravilu s končnico 'oba'). Tako, v celoti porabljeno besedo, lahko ločimo od ostalih ravno s to informacijo, da nima več nepregledanih končnic. Če se recimo nahajamo v vozlišču s končnico 'oba', ki vsebuje besede {'doba', 'grdoba', 'goba', 'oba'), ki imajo različne transformacije, potem lahko ločimo 3 izjeme, katerih končnice so ('doba', 'goba', '#oba'). Znak '#' predstavlja, da je beseda porabljena, zato so pravila s končnico '#' vedno listi RDR drevesa, saj ne morejo

imeti nobene izjeme. To informacijo lahko uporabimo tudi v osnovnem algoritmu, tako da vsem učnim in testnim primerom pripnemo nek unikatni znak, npr. '#', na začetek besede ter leme.

- Z originalnim algoritmom generirano drevo ima težave naštete v (3.3.5 Optimizacija). Če lahko te težave odpravimo že v učnem algoritmu, lahko izdelamo mnogo lepše drevo in na njemu ni potrebno izvajati postopka optimizacije.
- Želimo si tudi, da bi algoritem izdelal tako drevo, ki bi bilo na učnih primerih najboljše možno. Predvidevamo, da bi se s tem povečala točnost tudi na testnih primerih. Izpolnitev tega je neposredno povezana s prvima dvema točkama, saj izpolnitev teh dveh pomeni upoštevanje statistike in s tem najboljše možno drevo za učne podatke.
- V določenih primerih obstaja več izbir za transformacijo vozlišča kljub pogoju iz prejšnje točke. Npr. kadar imamo v vozlišču več enako močnih transformacij. V teh primerih bi želeli z upoštevanjem določenih hevristik izbrati tako transformacijo, ki poveča točnost drevesa na testnih podatkih.
- Povečanje učinkovitosti algoritma. Originalni algoritem je bil zaradi v prejšnjem poglavju opisane implementacijske težave, dokaj počasen pri večjih učnih množicah.

Sprva smo skušali vse razen zadnje zahteve izpolniti v okviru originalnega algoritma. To je izvedljivo vendar na račun še veliko slabše učinkovitosti. Učinkovitost pade zaradi RDR koncepta, ki je v svoji naravi inkrementalen in ne upošteva globalne statistike vozlišč. Zaradi tega se drevo med učenjem zelo spreminja, vozlišča spreminjajo svojo vlogo v hierarhiji in zamenjujejo privzete transformacije ter svoje izjeme.

4.4 PREKRIVNI RDR ALGORITEM

Odločili smo se torej za nov pristop k reševanju tega učnega problema. Naš algoritem ne deluje več po RDR konceptu, ampak po principu pokrivanja *(angl. covering)* primerov s pravili. Ker pa še vedno izdela RDR drevo, smo ga poimenovali prekrivni RDR algoritem.

4.4.1 OSNUTEK

Ideja je preprosta in temelji na združevanju besed z enako končnico. Če npr. v prvem koraku združimo vse besede s končnico 'a', lahko zanje izdelamo novo izjemo korenskega pravila. Za transformacijo izjeme vzamemo vedno tisto, ki je v tej skupini besed najpogostejša. Ker imamo verjetno v tej skupini še vedno veliko besed, jo razdelimo na podskupine s končnicami ('#a', 'aa', 'ba', …, 'za', 'ža'). V kolikor v določeni podskupini ni nobene besede, zanjo seveda ne bomo naredili izjeme. Prav tako ne naredimo izjeme, če imajo vse besede podskupine enako transformacijo kot nadrejena skupina, saj take pravilno klasificira že ta. To združevanje besed oz. drobljenje skupin izvajamo tako dolgo, dokler v vsaki skupini niso zgolj besede z enako transformacijo. Ustavimo se tudi takrat, kadar so znotraj neke skupine vse besede enake in jih med sabo ne moremo več ločiti.

Najpomembnejšo idejo za izredno učinkovito optimizacijo smo v podobni obliki že uporabili v poglavju 3.3.5 (Optimizacija). Gre za leksikografsko ureditev učnih primerov. Urejamo jih po morfoloških oblikah, vendar ne od začetka proti koncu besede, kot je v navadi, ampak obratno, po črkah nazaj. Obratno urejamo zato, ker je naša naloga združevati besede v skupine glede na njihovo končnico. Želimo npr., da v seznamu ležijo skupaj besede s končnico 'a', znotraj te skupine skupaj 'va' in tako rekurzivno naprej 'eva', 'ševa', 'iševa', 'piševa'. Tako ureditev omogoča ravno omenjeno obratno leksikografsko urejanje. Zgled urejenih besed vidimo v koloni *[Morf.]* iz primera 4.2.

4.4.2 IMPLEMENTACIJA

Koda 4.3 predstavlja naš učni algoritem. Zgornja tabela predstavlja zgolj okvir zaključene celote. Bistvo postopka je funkcija *BuildRDRTree*. Ta funkcija dobi kot parametra novo prazno pravilo in seznam primerov, ki naj jih to pravilo pravilno klasificira (lematizira). Opis kode 4.3 je naslednji:

- 2.2,2.3: Ker je *currRule* prazno pravilo, ga moramo najprej napolniti s podatki.
  - 2.2: Zanimivo pri naši ureditvi primerov je to, da lahko najdaljšo končnico, ki je enaka vsem besedam, dobimo enostavno s primerjanjem prve in zadnje besede iz seznama.
  - 2.3: Funkcija *MostFrequentTransformation* vrne najbolj pogosto transformacijo celotnega seznama. V kolikor je takih več, hevristično izberemo tisto izmed teh, ki je najbolj pogosta med vsemi besedami učne množice. Vendar pa mora biti transformacija taka, da jo je možno izvesti na vseh besedah seznama. Če imamo transformacijo *'from'->'to',* mora biti torej *'from'* krajši ali enak kot je končnica trenutnega primera.
  - 2.5,2.6: Če je končnica daljša od *'from'* dela transformacije, je to nepotrebno in jo skrajšamo na *'from'* oz. minimalno na dolžino končnice starša + 1.
- 2.9-2.12: Ustavitveni pogoj. Če so vsi primeri pravilno pokriti, zaključimo.
  - 2.10,2.11 Če je poleg tega transformacija enaka nadrejenemu vozlišču, lahko to izjemo preprosto izbrišemo, saj to ne spremeni klasifikacije, zmanjša pa drevo.
- 2.13, 2.14: Ustavitveni pogoj. Vse besede so enake, zato jih ne moremo več razdeliti.
- 2.22-2.33: Besede delimo v podskupine na podlagi končnice, ki je za 1 daljša od trenutne.
  - 2.23: Naredimo nov prazen seznam primerov podskupine.
  - 2.26-2.28: Dokler so končnice besed enake, polnimo seznam podskupine.
  - 2.31-2.33: Dodamo novo izjemo za ustvarjeno podskupino in rekurzivno pokličemo funkcijo *BuildRDRTree* s parametroma nove izjeme in nove podskupine.

KODA 4.3: PREKRIVNI RDR ALGORITEM

```
1.1  //pripravimo začetne podatke
1.2  rootRule = new empty rule
1.3  exampleList = all learning examples
1.4  SortReverse(exampleList)
1.5
1.6  //prožimo rekurzivno gradnjo drevesa
1.7  BuildRDRTree(rootRule, exampleList)
```

```
2.1  BuildRDRTree(currRule, exampleList)
2.2      //nastavi lastnosti trenutnega pravila (končnico in transformacijo)
2.3      currRule.suffix = LongestEqualSufix(exampleList.First(), exampleList.Last())
2.4      currRule.transformation = MostFrequentTransformation(exampleList)
2.5      if (suffix longer than 'from' part of transformation)
2.6         currRule.suffix = max(currRule.transformation.from, currRule.parent.suffix + 1)
2.7
2.8      //preveri ustavitvena pogoja (pravilna klasifikacija vseh besed ali konec besede)
2.9      if (all examples from exampleList have the same transformation)
2.10         if(currRule.transformation == currRule.parent.transformation)
2.11             delete currRule
2.12         return
2.13     if (exampleList.First() == exampleList.Last())
2.14         return
2.15
2.16     //pripravi spremenljivki za delitev besed v podskupine
2.17     example = exampleList.First()
2.18     subSuffixLen = Length(currRule.suffix) + 1
2.19
2.20     //razdeli besede v podskupine tako, da so v isti podskupini besede z končnico
2.21     //za ena daljšo, kot je končnica trenutnega pravila
2.22     while (example != exampleList.Last())
2.23         subExampleList = new empty list
2.24         subSuffix = Suffix(example , subSuffixLen)
2.25
2.26         while (subSuffix == Suffix(example , subSuffixLen))
2.27             subExampleList.Add(example)
2.28             example = exampleList.Next()
2.29
2.30         //dodamo izjemo
2.31         exceptionRule = new empty rule
2.32         currRule.AddException(exceptionRule)
2.33         BuildRDRTree(exceptionRule, subExampleList)
```

Omenimo še, da se v pravilu, kjer je končnica enaka besedi, še vedno po potrebi naredi izjema tega pravila. Ker mora biti končnica izjeme vsaj za 1 daljša od končnice pravila, zapišemo v tem primeru dodatni znak kot '#'. Ta znak (desetiška ascii vrednost je 35) smo uporabili tudi pri izvozu drevesa. Ker sta lematizator in generator lematizatorjev izdelana bolj splošno, je to edini način, kako informacijo o porabljeni celotni besedi posredujemo v ta dva sklopa. Edina negativna stran postopka je dodajanje '#' na začetek vsake besede med lematizacijo. Če tega ne storimo, bo lematizacija še vedno potekala pravilno, le te dodatne informacije ne bomo izkoriščali.

4.4.3 Časovna zahtevnost

Časovno zahtevnost algoritma je v najslabšem primeru *(angl. worst-case)* enostavno izračunati.

Iz psevdokode (koda 4.3) lahko razberemo, da mora postopek ob vsakem klicu funkcije *BuildRDRTree* dvakrat preleteti celotni seznam besed. V prvem izračuna funkcijo *MostFrequentTransformation* (vr. 2.4) ter preveri ustavitveni pogoj (vr 2.7). V drugem pa naredi podskupine za vse besede (vr. 2.20-2.31). Vsi ostali ukazi se izvršijo v konstantnem času.

Ker je postopek rekurziven, je resda na vsakem nivoju več klicev funkcije *BuildRDRTree,* vendar pa so tudi seznami besed krajši. Pomemben razmislek pa je, da se seznami vedno razdelijo tako, da se ne prekrivajo. Celotni seštevek velikosti vseh sezamov nekega nivoja tako ne more biti več kot velikost začetnega seznama. To se lepo vidi na primeru 4.2, kjer so nivoji rekurzije označeni kot dolžina končnice.

Določiti moramo še maksimalno število nivojev rekurzije. Ker se z vsakim novim klicem *BuildRDRTree,* končnica besede poveča minimalno za ena, je globina rekurzije omejena z najdaljšo besedo.

Če je N število učnih primerov ter M dolžina najdaljše besede, je časovna zahtevnost našega algoritma $O(2 * M * N)$. Če upoštevamo, da je dolžina najdaljše besede v večini jezikov omejena in torej konstantna, dobimo red zahtevnosti $O(N)$. Prekrivni RDR algoritem ima torej tudi v najslabšem primeru linearno časovno zahtevnost.

Izkaže se tudi, da v realnih problemih število primerov z vsakim nivojem hitro pada (primer 4.2), zato je konstantni faktor skoraj neodvisen od M ter zaradi dobre implementacije dokaj majhen.

4.5 Primera

4.5.1 Primerjava med algoritmoma

Primer 4.1 je naveden kot primerjava med starim in novim algoritmom. To drevo je zgrajeno na istih učnih primerih (označenih besedah primera 5.1) kot drevo iz primera 1.4. To smo zgradili z originalnim algoritmom. Vidimo, da se drevesi zaradi drugačnega koncepta gradnje precej razlikujeta.

Primer 4.1 pa lahko postavimo ob bok tudi primeru 2.1, ki predstavlja optimizirano drevo 1.4. Ker si nekaj enakih zahtev delita tako novi učni algoritem kot tudi optimizacijski postopek, lahko opazimo, da sta si drevesi precej bolj podobni. Na optimizacijo lahko zdaj pogledamo kot na postopek, kako pretvoriti poljubno drevo v obliko čim bolj podobno drevesu, ki ga vrne prekrivni RDR algoritem. Seveda se drevesa med optimizacijo vsebinsko ne spreminjajo. Če poskušamo optimizirati izhod

našega učnega algoritma, se drevo ne spremeni. Z optimizacijo dobimo torej nekakšno univerzalno predstavitev dreves, ki jih lahko med sabo nato enostavno vsebinsko primerjamo.

Vidimo, da je razlika med drevesi 2.1 in 4.1 le v dodanem vozlišču 1.1.1 pri slednjem. Ko preverimo učne podatke, vidimo, da smo uspeli iz podatkov izluščiti nekoliko več informacije. Za besedo 'pisali' je algoritem pravilno dodal novo izjemo. Postopek je upošteval statistiko in se odločil za transformacijo ('li'->'ti'), saj sta v njeno prid govorila dva primera in eden proti. Originalni pa je vzel prvo možnost, ki jo je videl ('i'->'o'). Naš algoritem je tako pravilno lematiziral eno besedo iz učne množice več (9/10) kot originalni (8/10).

PRIMER 4.1: RDR DREVO ZGRAJENO S PREKRIVNIM RDR ALGORITMOM

```
1       `---> RULE:( suffix("") transform(""-->"") except(3) ); {:
1.1           |---> RULE:( suffix("i") transform("i"-->"o") except(4) ); {:
1.1.1         |      |---> RULE:( suffix("li") transform("li"-->"ti") );
1.1.2         |      |---> RULE:( suffix("ni") transform("ni"-->"ti") );
1.1.3         |      |---> RULE:( suffix("ti") transform(""-->"") );
1.1.4         |      `---> RULE:( suffix("ši") transform("ši"-->"sati") ); :}
              |
1.2           |---> RULE:( suffix("l") transform("l"-->"ti") );
1.3           `---> RULE:( suffix("mo") transform(""-->"") except(2) ); {:
1.3.1               |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
1.3.2               `---> RULE:( suffix("šimo") transform("šimo"-->"sati") ); :}
                     :}
```

4.5.2 POTEK GRADNJE RDR DREVESA

Iz primera 4.2 lahko razberemo, kako poteka gradnja RDR drevesa z razvitim algoritmom.

V začetnih stolpcih so navedene besede, njihove leme ter ustrezne transformacije. Besede so urejene v zahtevanem vrstnem redu (glede na končnice). Stolpec *[Dolžina končnice]* si lahko predstavljamo tudi kot globino oz. nivo rekurzije. Algoritem potemtakem začne na nivoju 0 in nadaljuje po naraščajočih številkah. Bloki, predstavljeni z okvirji v tabeli, predstavljajo eno skupino besed oz. spremenljivko exampleList, hkrati pa tudi eno pravilo v drevesu. Besede iste skupine imajo enako končnico, katere dolžina je odvisna od nivoja.

Vsaka skupina oz. blok vsebuje 4 podatke:

- seznam besed, ki jih pokriva (pripadajoče besede desno od bloka),
- končnica pravila (prvi stolpec bloka),
- najboljša možna transformacija (drugi stolpec bloka),
- izjeme pravila (bloki levo od trenutnega, na katere se blok razdeli).

Barvna koda blokov je naslednja:

- rumena: standardno notranje vozlišče drevesa (pravilo z izjemami), ki se deli naprej

- zelena: končno vozlišče, ki pravilno lematizira vse svoje primere. Nadaljnja delitev ni potrebna.
- rdeča: končno vozlišče, ki ne lematizira pravilno vseh primerov, ampak, ker so si vse besede enake, se postopek zaključi zaradi pogoja (koda 4.3, vr. 2.13)
- bela: vmesno vozlišče, ki se ni generiralo, ker ne prinese nič dodatne informacije, saj skušamo narediti ravno prav dolge končnice (koda 4.3, vr. 2.5, 2.6).

Prazni bloki predstavljajo zaradi pogoja (koda 4.3, vr. 2.10, 2.11) eliminirana vozlišča, saj so njihove optimalne transformacije enake transformacijam njihovih nadrejenih vozlišč.

Začetek poteka gradnje za dani algoritem je naslednji:

- Vse besede se sortirajo in algoritmu predstavijo kot ena skupina (nivo 0). Algoritem najde najdaljšo skupno končnico '' in najboljšo transformacijo ''->''. Nato razdeli skupino na podskupine, katerih končnice so dolge 1, (nivo 1).
- Prva izjema, ki jo algoritem generira, ima končnico 'a' Zanjo najde najboljšo transformacijo 'a'->'o'. Išče izjeme dolžine 2.
- Prva izjema dolžine 2 ima končnico 'la' in najboljšo transformacijo 'a'->'o'. Vendar je enaka kot v nadrejenem vozlišču, zato se to vozlišče eliminira.
- Nadaljuje z izjemami nivoja 1. Naslednja izjema je 'ma'. Zanjo najde 'ma'->''. Ker ta pravilno pokriva obe besedi, skupine ne deli več.
- Spet nadaljuje na nivoju 1 v bloku 'a',…
- Ko izdela vse izjeme nivoja 1 za 'a', se vrne na nivo 0 in nadaljuje z izjemi dolžine 1, naslednja je 'e' potem 'h', 'i', …

Končni rezultat (RDR drevo) je prikazan na primeru 4.3.

PRIMER 4.2: DELOVANJE UČNEGA ALGORITMA

Št	Morf.	Lema	Transfor.	Dolžina končnice									
				0		1		2		3		4	
1	pisala	pisati	la->ti	-		a							
2	pisala	pisalo	a->o	-		a							
3	pisala	pisalo	a->o	-		a							
4	pisala	pisalo	a->o	-		a							
5	pisala	pisati	la->ti	-		a							
6	pisala	pisati	la->ti	-		a							
7	pisaloma	pisalo	ma->	-		a		ma	ma->				
8	pisaloma	pisalo	ma->	-		a	a->o	ma					
9	pisana	pisati	na->ti	-		a		na					
10	pisana	pisati	na->ti	-		a		na	na->ti				
11	pisana	pisati	na->ti	-		a		na					
12	pišeta	pisati	šeta->sati	-		a		ta		eta		šeta	
13	pišeta	pisati	šeta->sati	-		a		ta	a->o	eta	a->o	šeta	šeta->sati
14	pišita	pisati	šita->sati	-		a		ta		ita	a->o	šita	šita->sati
15	piševa	pisati	ševa->sati	-		a		va	a->o	eva	a->o	ševa	ševa->sati
16	pišiva	pisati	šiva->sati	-		a		va		iva	a->o	šiva	šiva->sati
17	pisale	pisati	le->ti	-		e		le	le->ti				
18	pisane	pisati	ne->ti	-		e		ne	ne->ti				
19	pišete	pisati	šete->sati	-		e		te		ete		šete	šete->sati
20	pišite	pisati	šite->sati	-		e	->	te	->	ite	->	šite	šite->sati
21	piše	pisati	še->sati	-		e		še	še->sati				
22	pisalih	pisalo	ih->o	-		h							
23	pisalih	pisalo	ih->o	-		h	ih->o						
24	pisali	pisati	li->ti	-		i		li					
25	pisali	pisati	li->ti	-		i		li					
26	pisali	pisati	li->ti	-		i		li					
27	pisali	pisalo	i->o	-	->	i		li	li->ti				
28	pisali	pisalo	i->o	-		i		li					
29	pisali	pisalo	i->o	-		i	i->o	li					
30	pisani	pisati	ni->ti	-		i		ni					
31	pisani	pisati	ni->ti	-		i		ni	ni->ti				
32	pisani	pisati	ni->ti	-		i		ni					
33	pisati	pisati	->	-		i		ti	->				
34	piši	pisati	ši->sati	-		i		ši	ši->sati				
35	pisal	pisati	l->ti	-		l							
36	pisal	pisalo	->o	-		l	->o						
37	pisal	pisalo	->o	-		l							
38	pišem	pisati	šem->sati	-		m		em	m->	šem	šem->sati		
39	pisalom	pisalo	m->	-		m	m->						
40	pisalom	pisalo	m->	-		m							
41	pisan	pisati	n->ti	-		n	n->ti						
42	pišejo	pisati	šejo->sati	-		o		jo	->	ejo	->	šejo	šejo->sati
43	pisalo	pisalo	->	-		o							
44	pisalo	pisati	lo->ti	-		o							
45	pisalo	pisalo	->	-		o	->						
46	pišemo	pisati	šemo->sati	-		o		mo		emo		šemo	šemo->sati
47	pišimo	pisati	šimo->sati	-		o		mo	->	imo	->	šimo	šimo->sati
48	pisano	pisati	no->ti	-		o		no	no->ti				
49	pisat	pisati	->i	-		t	->i						
50	pisalu	pisalo	u->o	-		u							
51	pisalu	pisalo	u->o	-		u	u->o						
52	pišeš	pisati	šeš->sati	-		š	->	eš	->	šeš	šeš->sati		

PRIMER 4.3: DREVO ZGRAJENO IZ BESED PRIMERA 4.2

```
RULE:( suffix("") transform(""-->"") except(11) ); {:
 |---> RULE:( suffix("a") transform("a"-->"o") except(4) ); {:
 |       |---> RULE:( suffix("ma") transform("ma"-->"") );
 |       |---> RULE:( suffix("na") transform("na"-->"ti") );
 |       |---> RULE:( suffix("ta") transform("a"-->"o") except(2) ); {:
 |       |       |---> RULE:( suffix("šeta") transform("šeta"-->"sati") );
 |       |       `---> RULE:( suffix("šita") transform("šita"-->"sati") ); :}
 |       |
 |       `---> RULE:( suffix("va") transform("a"-->"o") except(2) ); {:
 |               |---> RULE:( suffix("ševa") transform("ševa"-->"sati") );
 |               `---> RULE:( suffix("šiva") transform("šiva"-->"sati") ); :}
 |                :}
 |
 |---> RULE:( suffix("e") transform(""-->"") except(4) ); {:
 |       |---> RULE:( suffix("le") transform("le"-->"ti") );
 |       |---> RULE:( suffix("ne") transform("ne"-->"ti") );
 |       |---> RULE:( suffix("te") transform(""-->"") except(2) ); {:
 |       |       |---> RULE:( suffix("šete") transform("šete"-->"sati") );
 |       |       `---> RULE:( suffix("šite") transform("šite"-->"sati") ); :}
 |       |
 |       `---> RULE:( suffix("še") transform("še"-->"sati") ); :}
 |
 |---> RULE:( suffix("ih") transform("ih"-->"o") );
 |---> RULE:( suffix("i") transform("i"-->"o") except(4) ); {:
 |       |---> RULE:( suffix("li") transform("li"-->"ti") );
 |       |---> RULE:( suffix("ni") transform("ni"-->"ti") );
 |       |---> RULE:( suffix("ti") transform(""-->"") );
 |       `---> RULE:( suffix("ši") transform("ši"-->"sati") ); :}
 |
 |---> RULE:( suffix("l") transform(""-->"o") );
 |---> RULE:( suffix("m") transform("m"-->"") except(1) ); {:
 |       `---> RULE:( suffix("šem") transform("šem"-->"sati") ); :}
 |
 |---> RULE:( suffix("n") transform("n"-->"ti") );
 |---> RULE:( suffix("o") transform(""-->"") except(3) ); {:
 |       |---> RULE:( suffix("šejo") transform("šejo"-->"sati") );
 |       |---> RULE:( suffix("mo") transform(""-->"") except(2) ); {:
 |       |       |---> RULE:( suffix("šemo") transform("šemo"-->"sati") );
 |       |       `---> RULE:( suffix("šimo") transform("šimo"-->"sati") ); :}
 |       |
 |       `---> RULE:( suffix("no") transform("no"-->"ti") ); :}
 |
 |---> RULE:( suffix("t") transform(""-->"i") );
 |---> RULE:( suffix("u") transform("u"-->"o") );
 `---> RULE:( suffix("šeš") transform("šeš"-->"sati") ); :}
```

5 REZULTATI EKSPERIMENTOV

Poglavje podaja opis, rezultate ter interpretacijo dveh poskusov. V prvem smo naredili primerjavo med novim in obstoječim algoritmom glede točnosti učenja in hitrosti lematizacije. Eksperiment smo izvedli na več jezikih in tako dobili tudi oceno primernosti RDR lematizacije za različne jezike. V drugem poskusu pa je predstavljena aplikacija lematizacije, uporabljene kot stopnja predobdelave besedil in priprava za odkrivanje znanj v njih. Vzporednice naredimo na prikazu rezultatov odkrivanja znanj pred uporabo lematizacije in po njej.

5.1 UČENJE RDR PRAVIL NA LEKSIKONIH MULTEXT IN MULTEXT-EAST

V tem eksperimentu smo testirali:

- hitrost lematizatorja opisanega v poglavju 2,
- točnost, zanesljivost (standardno odstopanje) in hitrost učnega algoritma opisanega v poglavju 4,
- dovzetnost različnih jezikov na metodo RDR lematizacije.

Poleg omenjenega smo v poskusu izvedli še mnogo drugih implicitnih testov, ki jih tukaj ne navajamo. Zgled tega je npr. test identitete originalnega in optimiziranega lematizatorja, kar je bila zahteva v poglavju 3. Čeprav smo identičnost obeh lematizatorjev formalno oz. opisno že dokazali, smo tukaj naredili še eksperimentalni test. Preverili smo tudi mnogo ostalih zahtev za izdelane algoritme. Rezultatov ne podajamo, lahko pa povemo, da so bili na koncu vsi tovrstni testi uspešni.

Primerjave so narejene med naslednjimi algoritmi:

- Učenje RDR pravil:
  - Originalni algoritem iz [12], opisan v poglavju 4.2.
  - Novi algoritem razvit v tem delu, opisan v poglavju 4.4.
- Lematizacija:
  - Ad hoc metoda lematizacije opisana v prvi točki uvodnega razdelka poglavja 2.
  - Lematizator razvit v tem delu, njegovemu opisu pa je posvečeno jedro poglavja 2.

5.1.1 Opis podatkov

V testih smo uporabljali podatke iz korpusa člankov ter leksikonov Multext-East verzija 3 [5]. Iz te baze smo uporabili leksikone morfoloških oblik besed. Podmnožico zapisov iz leksikona podaja primer 5.1. Vsaka vrstica vsebuje po eno morfološko obliko besede. Prvi zapis v vrstici je oblika, naslednji lema besede in zadnji morfološka oznaka. Stolpci so ločeni z znakom tabulator *(angl. tab-delimited)*. Podatkov o morfoloških oznakah nismo uporabljali, zato tukaj niso natančneje razloženi, gre pa za oznake kot so besedna vrsta, spol, sklon, število, sklanjatev, itd.

Primer 5.1: Zapisi iz leksikona morfoloških oblik Multext-East
Iz leksikona so bile izbrane vse morfološke oblike dveh besedilnih enot. Samostalnik pisalo ima 18 oblik, glagol pisati pa 34. Obarvane besede so bile uporabljene za gradnjo primera RDR lematizacijskih pravil (glej primer 1.4).

1	piše	pisati	Vmip3s--n	27	pisali	pisati	Vmps-pma
2	pišejo	pisati	Vmip3p--n	28	pisalih	pisalo	Ncndl
3	pišem	pisati	Vmip1s--n	29	pisalih	pisalo	Ncnpl
4	pišemo	pisati	Vmip1p--n	30	pisalo	pisati	Vmps-sna
5	pišeta	pisati	Vmip2d--n	31	pisalo	pisalo	Ncnsn
6	pišeta	pisati	Vmip3d--n	32	pisalo	pisalo	Ncnsa
7	pišete	pisati	Vmip2p--n	33	pisalom	pisalo	Ncnpd
8	piševa	pisati	Vmip1d--n	34	pisalom	pisalo	Ncnsi
9	pišeš	pisati	Vmip2s--n	35	pisaloma	pisalo	Ncndd
10	piši	pisati	Vmmp2s	36	pisaloma	pisalo	Ncndi
11	pišimo	pisati	Vmmp1p	37	pisala	pisalo	Ncnpa
12	pišita	pisati	Vmmp2d	38	pisala	pisalo	Ncnpn
13	pišite	pisati	Vmmp2p	39	pisala	pisalo	Ncnsg
14	pišiva	pisati	Vmmp1d	40	pisalu	pisalo	Ncnsd
15	pisal	pisati	Vmps-sma	41	pisalu	pisalo	Ncnsl
16	pisal	pisalo	Ncndg	42	pisan	pisati	Vmp--smp
17	pisal	pisalo	Ncnpg	43	pisana	pisati	Vmp--dmp
18	pisala	pisati	Vmps-dma	44	pisana	pisati	Vmp--pnp
19	pisala	pisati	Vmps-pna	45	pisana	pisati	Vmp--sfp
20	pisala	pisati	Vmps-sfa	46	pisane	pisati	Vmp--pfp
21	pisale	pisati	Vmps-pfa	47	pisani	pisati	Vmp--dfp
22	pisali	pisalo	Ncnda	48	pisani	pisati	Vmp--dnp
23	pisali	pisalo	Ncndn	49	pisani	pisati	Vmp--pmp
24	pisali	pisalo	Ncnpi	50	pisano	pisati	Vmp--snp
25	pisali	pisati	Vmps-dfa	51	pisat	pisati	Vmu
26	pisali	pisati	Vmps-dna	52	pisati	pisati	Vmn

Multext-East zbirka vsebuje leksikone v naslednjih jezikih: slovenski, srbski, bolgarski, češki, angleški, estonski, francoski, madžarski in romunski.

Poleg te zbirke smo po končanem razvojnem delu te diplomske naloge prejeli še podatke za nekaj jezikov iz osnovnega projekta Multext *(Multilingual Text Tools and Corpora)*[7]. Dodatni jeziki iz te zbirke so: angleški, francoski, nemški, italijanski in španski. Med skupne rezultate smo vključili tudi te jezike, zaradi nazornejše predstavitve smo jih obdržali skupaj v ločeni skupini označeni z *MULTEXT*. Osnovni podatki so označeni z *MULTEXT-EAST*. Skupaj imamo tako 14 leksikonov za 12 različnih jezikov.

V tabeli 5.1 smo podali nekaj osnovnih statistik leksikonov. Zanimiva sta predvsem zadnja dva podatka. Prvi posredno prikazuje kompleksnost jezika. Lahko bi rekli, da več kot ima vsaka lema (besedilna enota) različnih morfoloških oblik (sklanjatev, sklonov, števil, ..), težji je jezik. Ob tem je potrebno upoštevati tudi ostale podatke, npr. število vseh zapisov, ki prikaže tudi kakovost podatkov za določeni jezik. Vendar v našem primeru ta podatek niti ni tako zaskrbljujoč kot podatek o tem, koliko različnih lem ima v povprečju ena morfološka oblika. Problem našega načina lematizacije je namreč ta, da v primeru, ko ima ena oblika več lem, ne znamo nedvoumno določiti prave leme tej obliki.

TABELA 5.1: OSNOVNE STATISTIKE LEKSIKONOV IZ BAZ MULTEXT-EAST TER MULTEXT

	jezik	število zapisov	število različnih				
			morf. oblik	lem	morf. oznak	morf. oblik za isto lemo	lem za isto morf obliko
MULTEXT- EAST	slovenski	557.970	198.507	16.389	2.083	12,63	1,0430
	srbski	20.294	16.907	8.392	906	2,07	1,0285
	bolgarski	55.200	40.910	22.982	338	1,95	1,1002
	češki	184.628	57.391	23.435	1.428	2,55	1,0441
	angleški	71.784	48.460	27.467	135	1,80	1,0206
	estonski	135.094	89.591	46.933	643	2,19	1,1507
	francoski	306.795	232.079	29.446	380	8,01	1,0164
	madžarski	64.042	51.095	28.090	619	2,03	1,1209
	romunski	428.194	352.279	39.359	616	9,35	1,0447
MULTEXT	angleški	66.216	43.371	22.874	133	1,93	1,0182
	francoski	306.795	232.079	29.446	380	8,01	1,0164
	nemški	233.858	51.010	10.655	227	4,87	1,0174
	italijanski	145.530	115.614	8.877	247	13,85	1,0636
	španski	510.709	474.158	13.236	264	36,07	1,0069

Četudi v tem trenutku ne bi vedeli še ničesar o rezultatih, lahko na podlagi povedanega že sklepamo na nekaj zaključkov. Jeziki z veliko različnih lem za isto morfološko obliko (estonski, madžarski, bolgarski) bodo imeli verjetno slabo točnost lematizacije. To se bo pokazalo že pri učni množici, saj se enostavno ne moremo odločiti, katero lemo naj pripišemo besedi, če jih ima le ta več. Po drugi strani pa pričakujemo, da bo jezik z najmanj lem na morfološko obliko (španski) dobro lematiziran ne glede na njegovo oblikovno pestrost (v povprečju ima kar 36 različnih morf. oblik za vsako lemo). Podobno predvidevamo tudi za ostale jezike z malo lem na morf. obliko (francoski, nemški, angleški). Pri ocenah moramo upoštevati tudi kvaliteto podatkov: kako dobro in koliko besed je v leksikonu glede na vse besede jezika. Najbolj negotov se v tem primeru zdi srbski jezik, ki ima kljub svoji pestrosti (št. različnih morfoloških oznak) izredno malo besed v leksikonu. Morda lahko tudi zaradi tega pričakujemo slabše rezultate.

5.1.2 REZULTATI

Poskuse smo izvajali po metodi prečnega preverjanja K-tega reda *(angl. K-fold cross validation)*. Za K smo po zgledu članka [13] izbrali vrednost 5. Vsakič, ko smo naredili novo prečno preverjanje, smo torej celotno množico vseh primerov razbili na pet podmnožic. Štiri izmed teh smo uporabili za učenje, eno pa za test točnosti. Tako smo v vsakem preverjanju naredili dejansko 5 primerjav. Ker

smo želeli dobiti čim bolj zanesljivo oceno za standardno odstopanje, smo celotno prečno preverjanje izvedli za vsak jezik 10 krat. Poleg tega imamo dva algoritma in dve možni razbitji na testno in učno množico, kar doda nov faktor 4. Za vsak jezik smo tako zgradili $5 * 20 * 4 = 400$ različnih dreves. Če upoštevamo, da imamo poleg tega še 14 različnih leksikonov, pridemo do številke $14 * 400 = 5600$ zgrajenih dreves.

Tabele 5.2, 5.3 in 5.4 prikazujejo rezultate poskusov. Interpretacije rezultatov sledijo v naslednjem poglavju, tukaj pa podajamo določene ne trivialne podatke oz. nazive stolpcev tabel. V vseh tabelah najdemo naziva *[rdr]* oz. *[cover]*, ki se nanašata na originalni RDR oz. novi prekrivni RDR algoritem. V kolikor govorimo o učenju, mislimo na učni algoritem sicer pa na algoritem lematizacije. Ostali podatki so glede na tabele naslednji:

- Tabela 5.2:
  - *[točnost]* – ocena točnosti lematizacije. Če predpostavimo, da je lematizacija pravzaprav samo klasifikacija morfološke oblike besede v pravi razred (transformacijo), potem lahko temu podatku rečemo tudi klasifikacijska točnost. Če je N_p število pravilno lematiziranih besed in N število vseh besed, potem točnost izračunamo po enačbi $T = \frac{N_p}{N} * 100\%$.
  - *[učna množica (optmistično)]* – za izračun ocen smo uporabili podatke iz učne množice in tako dobili optimistično oz. najboljšo možno vrednost.
  - *[testna množica (prib. realno)]* – za izračun ocen so bili uporabljeni podatki testne množice. To je približno realna ocena vrednosti točnosti.
  - *[*neznane besede (pesimistično)*]* - za izračun ocen so bili prav tako uporabljeni podatki iz testne množice, vendar je bilo razbitje testne in učne množice bolj striktno. Poskrbeli smo, da se dve besedi, ki imata enako morfološko obliko in lemo, nikoli ne pojavita hkrati v učni in testni množici. Več o tem, kaj ta podatek pravzaprav pomeni, je razloženo v sledeči interpretaciji rezultatov.
  - *[standardno odstopanje]* – ocena odstopanja točnosti T enega poskusa od povprečne točnosti $\bar{T}$ skozi vse poskuse. Nižje vrednosti kažejo na večjo stabilnost učnega algoritma. Izračunamo ga po enačbi $s = \sqrt{\frac{\sum_{i=1}^{K}(T_i - \bar{T})^2}{K-1}}$, kjer je K število izvedenih poskusov. Za točnost smo vzeli podatke vseh treh točnosti.
  - *[napak]* – relativno zmanjšanje števila napak prekrivnega algoritma proti RDR. Enačba za *[napak]* točnosti je $d_T = \frac{(T_{orig.} - T_{novi})}{(1 - T_{orig.})}$. Podobno definiramo tudi *[napak]* standardnega odstopanja. Če je *[napak]* recimo -25, to pomeni, da prekrivni algoritem naredi 25% manj napak kot RDR.

TABELA 5.2: PRIMERJAVA TOČNOSTI IN STANDARDNEGA ODSTOPANJA UČNIH ALGORITMOV

	jezik	točnost (%)									standardno odstopanje (%)		
		učna množica (optimistično)			testna množica (prib. realno)			neznane besede (pesimistično)					
		rdr	cover	napak	rdr	cover	napak	rdr	cover	napak	rdr	cover	napak
MULTEXT- EAST	slovenski	95,35	97,61	-48,6	92,59	94,38	-24,1	80,68	82,12	-7,5	0,029	0,015	-47,88
	srbski	94,36	97,86	-62,1	70,34	73,49	-10,6	64,26	65,85	-4,5	0,150	0,059	-60,44
	bolgarski	91,22	93,68	-28,0	74,52	76,10	-6,2	69,29	71,52	-7,2	0,107	0,074	-30,29
	češki	96,61	97,89	-37,8	92,77	93,66	-12,3	78,09	81,13	-13,9	0,040	0,023	-41,02
	angleški	97,75	98,84	-48,3	92,05	93,07	-12,8	89,27	91,03	-16,4	0,038	0,021	-45,27
	estonski	86,81	89,51	-20,5	73,52	73,93	-1,6	66,69	66,54	0,5	0,066	0,049	-25,83
	francoski	96,72	98,80	-63,5	91,78	92,94	-14,1	86,80	88,22	-10,8	0,032	0,015	-54,19
	madžarski	90,23	91,88	-16,9	74,82	74,33	2,0	72,73	72,86	-0,5	0,091	0,072	-21,03
	romunski	94,96	96,75	-35,6	78,16	79,17	-4,6	73,48	74,14	-2,5	0,036	0,033	-7,27
MULTEXT	angleški	98,20	99,00	-44,5	93,29	94,14	-12,7	90,82	92,48	-18,1	0,052	0,029	-45,17
	francoski	96,72	98,80	-63,5	91,79	92,95	-14,2	86,85	88,25	-10,7	0,034	0,012	-63,71
	nemški	95,88	98,70	-68,5	95,06	97,13	-41,9	79,56	84,15	-22,4	0,062	0,026	-58,54
	italijanski	93,75	95,58	-29,2	85,87	86,08	-1,5	82,05	82,11	-0,3	0,041	0,040	-3,26
	španski	99,10	99,48	-42,1	94,65	95,73	-20,1	94,32	95,45	-19,9	0,007	0,008	7,42

- Tabela 5.3:
  - *[ms/zap]* – število milisekund (10^{-6}s), ki jih porabi povprečno učni algoritem za učenje enega zapisa iz učne množice.
  - *[ns/zap]* - število nanosekund (10^{-9}s), ki jih porabi povprečno algoritem za lematizacijo ene besede iz testne množice.
  - *[faktor]* – faktor povečanja hitrosti novega učnega algoritma v primerjavi s starim.

TABELA 5.3: PRIMERJAVA HITROSTI UČNIH TER LEMATIZACIJSKIH ALGORITMOV

	jezik	učenje					lematizacija				
		rdr		cover		faktor	rdr		cover		faktor
		sek.	ms/zap.	sek.	ms/zap.		sek.	ns/zap.	sek.	ns/zap.	
MULTEXT- EAST	slovenski	26,80	60,0	3,02	6,8	8,9	2,53	22.633	0,10	867	26,1
	srbski	0,23	14,4	0,09	5,4	2,7	0,03	8.089	0,00	643	12,6
	bolgarski	2,03	46,0	0,32	7,2	6,4	0,30	26.958	0,01	670	40,2
	češki	4,42	29,9	0,56	3,8	7,9	0,42	11.279	0,03	722	15,6
	angleški	0,43	7,5	0,23	4,0	1,9	0,10	6.946	0,01	752	9,2
	estonski	4,15	38,4	0,68	6,3	6,1	0,41	15.226	0,02	800	19,0
	francoski	6,46	26,3	1,72	7,0	3,7	1,35	21.995	0,06	898	24,5
	madžarski	0,99	19,4	0,23	4,5	4,3	0,12	9.575	0,01	718	13,3
	romunski	183,12	534,6	7,23	21,1	25,3	43,51	508.043	0,08	911	557,7
MULTEXT	angleški	0,37	6,9	0,21	3,9	1,8	0,08	6.017	0,01	724	8,3
	francoski	7,02	28,6	1,56	6,3	4,5	1,34	21.819	0,05	877	24,9
	nemški	10,22	54,6	0,80	4,3	12,9	0,60	12.857	0,04	788	16,3
	italijanski	1,18	10,1	0,80	6,9	1,5	0,26	8.821	0,03	860	10,3
	španski	22,97	56,2	3,88	9,5	5,9	3,57	34.923	0,09	894	39,1

- Tabela 5.4:
  - *[število vozlišč]* – število vseh vozlišč (pravil) RDR drevesa.
  - *[povpr. globina]* – povprečna globina drevesa glede na razporeditev vozlišč. Pri tem podatku je treba biti pozoren, saj to ne pomeni, da bo lematizator v povprečju naredil toliko skokov po drevesu. Med lematizacijo se lahko izkaže, da se večino

proženih pravil nahaja v listih oz. blizu korena drevesa. To je odvisno tudi od podatkov, ki jih lematiziramo.

- *[povpr. vejanje]* – kolikšno je povprečno vejanje (število izjem) za notranja vozlišča drevesa.
- *[listi/notranja]* – razmerje med številom listov drevesa (pravila brez izjem) in številom notranjih vozlišč (pravila z izjemami).

TABELA 5.4: PRIMERJAVA VELIKOSTI IN LASTNOSTI IZDELANIH RDR DREVES

	jezik	število zapisov	število vozlišč		povpr. globina		povpr. vejanje		listi/notranja	
			rdr	cover	rdr	cover	rdr	cover	rdr	cover
MULTEXT- EAST	slovenski	557.970	32562	33643	8,0	14,0	5,3	3,0	4,26	1,98
	srbski	20.294	4941	4933	6,9	8,0	4,0	3,0	3,01	1,96
	bolgarski	55.200	8335	8453	6,6	9,5	4,6	3,1	3,63	2,08
	češki	184.628	11968	11239	6,1	10,0	5,2	3,2	4,17	2,22
	angleški	71.784	4421	4007	6,9	9,2	4,7	2,7	3,70	1,72
	estonski	135.094	17055	17480	8,3	11,9	4,4	3,0	3,44	2,05
	francoski	306.795	20446	22093	9,0	12,4	4,6	2,9	3,63	1,94
	madžarski	64.042	7723	8209	7,5	10,2	4,4	3,0	3,41	1,98
	romunski	428.194	66809	79303	9,0	14,3	7,0	2,8	6,04	1,81
MULTEXT	angleški	66.216	3631	3123	6,1	8,9	4,8	2,8	3,82	1,79
	francoski	306.795	20442	22102	9,0	12,4	4,6	2,9	3,63	1,94
	nemški	233.858	9032	9431	7,0	11,0	5,5	3,1	4,53	2,07
	italijanski	145.530	10891	13279	7,8	12,8	4,1	2,8	3,12	1,80
	španski	510.709	25588	21264	9,3	13,0	6,7	2,8	5,67	1,83

5.1.3 INTERPRETACIJA

Iz tabele 5.2 razberemo, da je prekrivni RDR algoritem v večini primerov boljši od originalnega. Temu veliko prispeva dejstvo, da smo skušali iz leksikonov izluščiti čim več podatkov, ki jih RDR algoritem ni upošteval. Tu je mišljena predvsem statistika besed in ostali principi opisani v poglavju 4.3 "Predlagane izboljšave".

Uporaba statistike največ pripomore k dvigu točnosti pri testih na učni množici (na besedah, ki jih je učni algoritem že videl). Izkušenim strokovnjakom iz področja strojnega učenja se bo zdel ta podatek morda zanemarljiv, saj nas redkokdaj zanima točnost na učni množici. V domeni lematizacije temu ni tako, saj imamo opravka z zelo popolno množico učnih primerov. Učni primeri tako pokrijejo večino domene in pri dejanski lematizaciji obstaja velika verjetnost, da smo se leme že naučili. Ob pregledu, koliko celotnega prostora domene prekrivajo naši učni primeri, smo naredili test s članki opisanimi v 5.2 ter leksikonom za slovenski jezik. Rezultat pokritosti je bil cca. 84%. To pomeni, da v 84% primerih lematizacije besed iz teh člankov pričakujemo točnost iz učne množice (optimistično), v ostalih 16% pa točnost iz neznanih besed (pesimistično). Najbolj realna ocena, ki jo dobimo po tej predpostavki, je torej $T = p * T_{optim.} + (1 - p) * T_{pesim.}$, kjer p predstavlja odstotek pokritosti besedil z besedami iz učne množice. Ta test torej zahteva referenčno besedilo, zato smo ga naredili samo za slovenščino, dobljena točnost je $T_{slo\_real} = 84\% * 97{,}61\% + 16\% * 82{,}12\% = 95.13\%$,

kar je še boljše kot naša ocena na testni množici. Seveda pa je odvisna od besedila in verjetno bi bila precej nižja, če bi lematizirali npr. znanstvene članke.

Opazimo tudi, da se je točnost povečala v večini jezikov pri vseh opazovanih merah. Zmanjšanje standardnega odstopanja interpretiramo kot povečanje zanesljivosti učenja. Tako točnost dejanskega končnega lematizatorja v povprečju leži bližje točnosti izračunani med testi prečnega preverjanja.

V tabeli 5.2 lahko analiziramo tudi primerjavo med različnimi jeziki. Točnosti se precej dobro držijo napovedi iz poglavja 5.1.1. Izstopajoča rezultata nam predstavljata madžarščina in estonščina, saj se izkaže, da je RDR algoritem pod določenimi pogoji celo boljši od prekrivnega. Med analizo tega pojava smo odkrili, da madžarščina in estonščina ne spadata med indo-evropske jezike ampak med ugrofinske. V to skupino se med evropskimi jeziki uvršča še finščina, morfološko gledano pa spadajo med aglutinativne jezike. Zanje velja, da lahko besede sestavljamo iz posameznih morfemov (osnovnih pomenskih enot) na veliko možnih načinov. V madžarščini ima lahko npr. en samostalnik teoretično tudi več milijonov različnih oblik. Za te jezike očitno celotni koncept zamenjevanja končnic pri lematizaciji ne deluje prav dobro.

Tabele 5.3 ne bomo podrobneje razlagali, saj je očitno razvidno ne le bistveno hitrejše izvajanje učnega ter lematizacijskega algoritma temveč tudi druga časovna zahtevnost algoritmov. To se vidi s primerjavo faktorjev razlik pri naraščajočih učnih množicah. V kolikor bi bila algoritma istega časovnega reda, bi bil faktor vedno enak. Zanimiv je še pogled na hitrost *[ns/zap.]* lematizacije prekrivnega algoritma. Ta je skoraj konstanta in neodvisna od velikosti drevesa, kar je zelo zaželeno.

Razlago tabele 5.3 zahteva edino rezultat za romunščino. Tu sta se oba algoritma izkazala zelo slabo. Pri pregledu podatkov smo ugotovili znaten delež besed, katerim se med spreganjem ne menja zgolj končnica, ampak tudi začetek. Zaradi tega so se drevesa zelo napihnila, kar kaže tudi tabela 5.4.

Tabeli 5.4 podaja primerjavo med algoritmoma glede na obliko izdelanih RDR dreves. Najprej opazimo, da so drevesa prekrivnega algoritma malenkost večja, predvsem pa globlja in manj razvejana. Rezultate enostavno pojasnimo s tem, da prekrivni algoritem izdela bolj uravnotežena drevesa, medtem ko imajo nekatera vozlišča drevesa algoritma RDR lahko tudi po 800 in več vozlišč. To veliko vejanje v začetnih vozliščih je sicer nezaželeno in je glavni razlog za take rezultate.

Ko naredimo povzetek tabel, ugotovimo, da je izboljšava novega prekrivnega algoritma tem boljša, čim boljši so učni podatki. To nam govori že dejstvo, da so največje razlike med algoritmoma ravno pri testu na učni množici. Ker se ukvarjamo z končno in razmeroma dobro pokrito domeno, je ta podatek pomemben. Vendar pa je tudi možnost učenja v smislu posploševanja informacije večja, kar nam kažejo testi na neznanih besedah. Zaključujemo, da je v veliki večini primerov bolje (glede hitrosti in točnosti) uporabljati nov prekrivni algoritem. Pokazateljev kdaj bi bilo bolje uporabljati izhodiščni RDR algoritem nismo našli.

5.2 APLIKACIJA NA AGENCIJSKIH ČLANKIH

Ta aplikacija je predstavlja naš motiv za izboljšavo lematizatorja v diplomskem delu. V nadaljevanju po opisu podatkov pokažemo dva primera ontologij. Prva ontologija je bila zgrajena iz ne-lematiziranih besedil, druga pa iz lematiziranih. Tako na primeru pokažemo resnični pomen lematizacije in njeno vlogo v okviru predobdelave besedil za potrebe odkrivanja znanj. V zaključku poglavja se dotaknemo še nekaj zanimivih zgledov ontologij, ki jih navajamo v prilogi C.

5.2.1 OPIS PODATKOV

Podatke nam je posredoval dr. Mihael Kline iz podjetja Kline&Kline in predstavljajo bazo člankov oz. novic o notranjih dogodkih v Sloveniji. Članki so povzeti od tiskovnih agencij različnih držav, v katerih vlada nek minimalni prag zanimanja za dogodke pri nas. Naloga tiskovne agencije je spremljati dogodke po svetu in poročati o stvareh, ki so zanimive za lokalno populacijo. V nadaljevanju te članke imenujemo kar agencijski članki.

Novice povzete od tujih tiskovnih agencij so seveda v njihovem jeziku, vendar smo mi dobili že prevedena in do določene mere okrajšana besedila. V teh povzetkih so večinoma ohranjene le najbistvenejše informacije iz originalnih novic. Članke smo predhodno obdelali, spravili v enotno obliko in združili po državi izvora. Kadar rečemo npr. avstrijski članki, mislimo torej na povzetke novic, ki jih je objavila avstrijska tiskovna agencija o Sloveniji. Nekatere države imajo tudi več tiskovnih agencij.

Naša naloga je bila narediti nekakšen pregled aktualnih dogodkov, o katerih se poroča v različnih državah. Glede na poročila se oblikuje tudi javno mnenje državljanov, zato bi tako dobili precej dobro informacijo o tem, kakšen je izgled Slovenije v očeh tujcev. Zanimala pa nas je tudi primerjava vsebin različnih jezikov med sabo in kot poseben primer, primerjava vsebin tujih jezikov s slovenskimi.

Slovenskih člankov je bila velika večina, kar je popolnoma razumljivo, saj Slovenska tiskovna agencija (STA) poroča večinoma o vseh pomembnih dogodkih v Sloveniji. Tabela 5.5 podaja število člankov za vse države, ki so se pojavile v podatkih. Tu velja omeniti, da se v istem članku lahko pojavi tudi več agencij, zato je bilo skupnih člankov za tuje agencije samo 2711 in ne 4081 kot bi pričakovali iz tabele 5.5. Prekrivanje tujih člankov je bilo tako 1,5 kratno. Tabela 5.5 je v bistvu že sama po sebi zanimiv rezultat, saj kaže stopnjo zanimanja tujih držav za nas. Datumsko se novice nanašajo na čas od 1.1.2006 do 23.2.2006, kar je pomembno, saj so tematike temu ustrezne.

TABELA 5.5: PRIMERJAVA ŠTEVILA AGENCIJSKIH NOVIC PO DRŽAVAH

država	tiskovne agencije	št. člankov
Slovenija	STA	18676
Srbija in Črna Gora	Beta, Mina, Tanjug	1562
Hrvaška	HINA	998
Avstrija	APA	611
ZDA	APA	213
BIH	FENA	179
Francija	AFP	170
Nemčija	DPA	156
Makedonija	MIA, MAKFAX	100
Rusija	ITAR-TAS	43
Italija	ANSA	21
Madžarska	MTI	14
Slovaška	TASR	9
Ciper	CNA	3
Azerbajdžan	AzerTac	2

Povprečna dolžina novice je 42 besed oz. 335 znakov. V primeru 5.2 je podan zgled dveh tipičnih novic iz našega korpusa. Zgornji članek je dobljen iz podatkov STA, spodnji pa je zgled iz tujih virov. Vidimo lahko precejšnjo razliko med članki STA in tujimi, saj gre pri slednjih zgolj za povzetke o tem, kar so agencije dejansko poročale.

PRIMER 5.2: DVE TIPIČNI NOVICI

1	LJUBLJANA - Najmočnejši pečat s svojo osebnostjo in dosežki je v minulem letu po mnenju uredniškega kolegija časnika Delo vtisnil filozof, prevajalec, publicist in pesnik Gorazd Kocijančič, ki si je prislužil naziv Delova osebnost leta. Osebnost leta je na slovesnosti v Muzeju novejše zgodovine v Ljubljani razglasil odgovorni urednik Dela Darijan Košir.
2	Ameriška tiskovna agencija AP je poročala, da slovenski zunanji minister Dimitrij Rupel odhaja kot novi predsedujoči Organizaciji za varnost in sodelovanje v Evropi (OVSE) v torek na dvodnevni delovni obisk v Ukrajino. Rupel se bo tam sestal s predsedniškima kandidatoma Viktorjem Juščenkom, ki je zmagal na ponovljenem drugem krogu volitev 26. decembra, in Viktorjem Janukovičem ter z odhajajočim predsednikom Leonidom Kučmo, je tudi poročala AP.

Predobdelava podatkov je bila zelo zahtevna, saj so članki prihajali iz različnih virov in so bili zato v različnih formatih. Večinoma jih je bilo v tekstovnih datoteki in datotekah programa *Microsoft Word*. Tudi formati posameznih člankov so bili zelo različni, saj so nekateri celo vsebovali glave elektronskih pošt ljudi, ki so si jih pošiljali. Tako je bilo za urejanje potrebnega tudi veliko ročnega dela. Vse novice pa so imele nekaj metapodatkov npr. datum objave, inicialke poročevalca, klasifikacijo po tematiki in nekaj drugih. Med postopkom predobdelave smo metapodatke ohranili, čeprav se je kasneje izkazalo, da jih ne uporabljamo.

Pred seboj smo imeli problem, kako iz nekaj tisoč novic izluščiti glavne tematike, ki v njih nastopajo. Odločili smo se, da celotno množico člankov posamezne države predstavimo z eno ontologijo [6]. Ontologija je sicer splošno v SSKJ definirana kot: "filozofska disciplina, ki obravnava bistvo in najsplošnejše lastnosti stvarnosti". V našem primeru pa gre za hierarhično strukturo, ki je

urejena glede na splošnost/specifičnost določenih vsebin. Ontologija, ki predstavlja novice, ima tako v korenu neko predstavitev (ključne besede), ki ustrezajo vsem novicam. Ta se nato razdeli na glavne teme, recimo gospodarstvo, politika, šport, kultura, …, katere se naprej delijo glede na vsebino obravnavanih novic. S pregledom take hierarhične razdelitve vsebin oz. tematik lahko hitro ugotovimo o čem članki v splošnem govorijo.

5.2.2 GRADNJA ONTOLOGIJ

Gradnjo ontologij nam je omogočil odlični, za to namenjeni sistem, imenovan Ontogen [6]. Avtorji so opredelili Ontogen kot pol avtomatični, podatkovno voden sistem za gradnjo preprostih ontologij, s katerim lahko hierarhično razbijemo koncept na podkoncepte opisane s ključnimi besedami, edina relacija med njimi pa je "podkoncept" *(angl. subconcept of)*. Delo z njim je preprosto, saj kot vhod vzame kar zbirko člankov in potem le z malo pomoči uporabnika generira dokaj dobro ontologijo. V kolikor mu posvetimo več časa, pa lahko z njegovo pomočjo zgradimo precej dobre ontologije. Nekaj takih ontologij je prikazanih tudi v prilogi C.

Zaradi narave algoritmov, ki jih uporablja Ontogen (algoritmi odkrivanja znanj iz besedil za razvrščanje dokumentov v skupine), pa je obvezno, da besedila lematiziramo, v kolikor želimo dobiti dobre rezultate. Ontogen že sam ponuja možnost lematizacije, a na žalost med jeziki še ni slovenščine. Tako nam ni preostalo drugega, kot da besedila lematiziramo pred uvozov v Ontogen. V kolikor lematizacije ne naredimo, nam kljub vložene veliko energije ne uspe zgraditi lepe ontologije. Tak primer prikazuje diagram 5.1, ki je zgrajen na vseh slovenskih STA člankih, a brez uporabe lematizacije. Tu je potrebno dodati še, da besede, ki so napisane za predstavitev konceptov, generira Ontogen avtomatično in jih nismo popravljali v nobeni prikazani ontologiji. Predstavljajo pa najmočnejše ključne besede posameznih konceptov.

V diagramu 5.2 lahko vidimo ontologijo generirano iz istih podatkov vendar na podlagi lematiziranih besedil. Tu se jasno vidi delitev tematike: vlada in ministrstva, gospodarstvo, evropska unija, državni zbor in zakoni ter sociala. Zametke te razdelitve lahko sicer razberemo tudi iz diagrama 5.1, a je tukaj prikaz bolj zamegljen. Še večje razlike opazimo na naslednjem nivoju, saj imajo koncepti iz lematiziranega diagrama precej dobro opredelitev o čem govorijo, iz nelematiziranega pa le tu in tam.

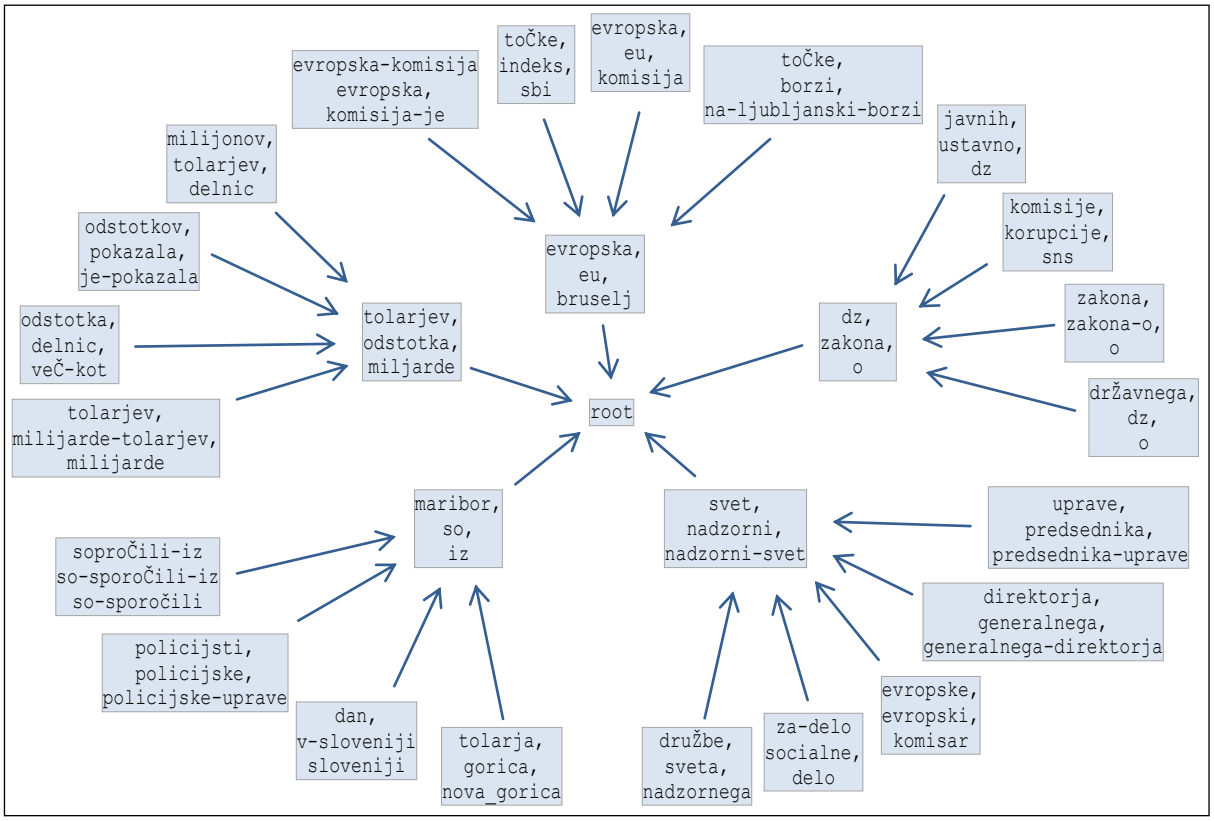


DIAGRAM 5.1: ONTOLOGIJA IZDELANA IZ LEMATIZIRANIH BESEDIL SLOVENSKIH AGENCIJSKIH ČLANKOV

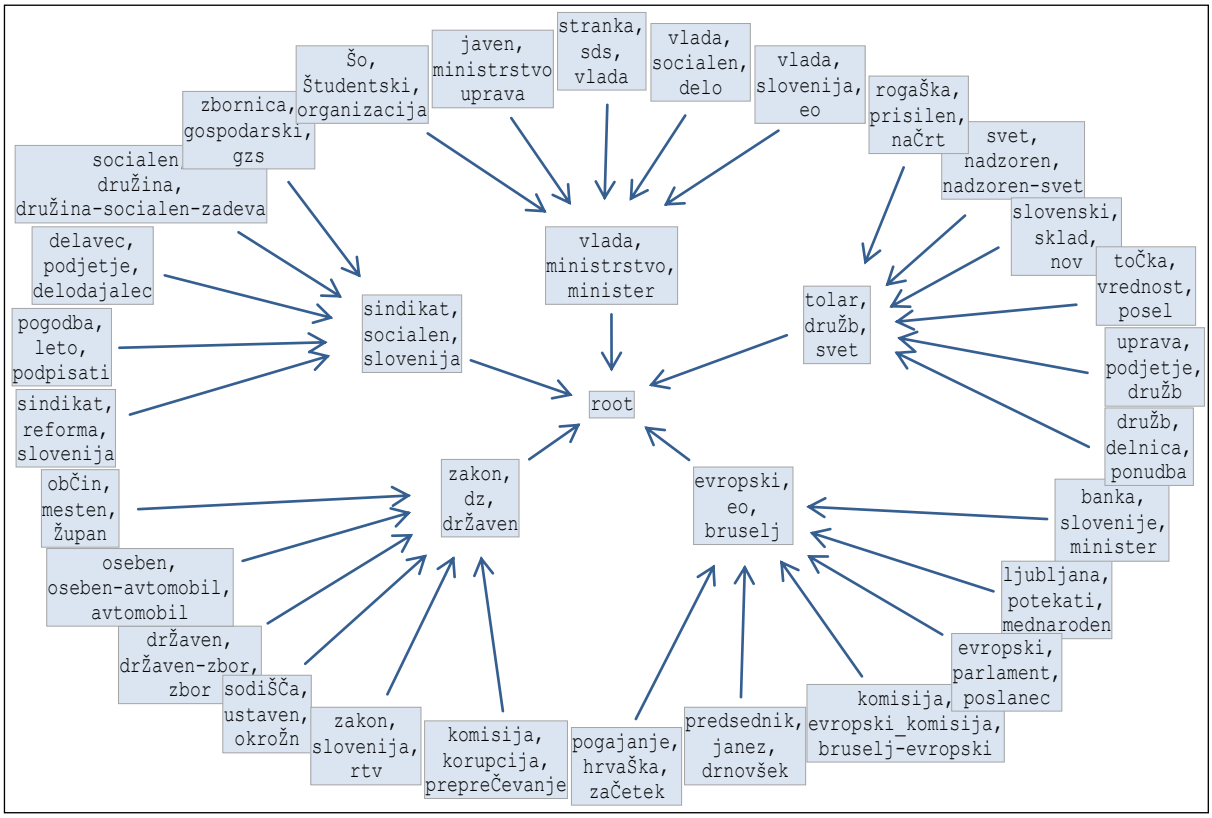


DIAGRAM 5.2: ONTOLOGIJA IZDELANA IZ NELEMATIZIRANIH BESEDIL SLOVENSKIH AGENCIJSKIH ČLANKOV

5.2.3 KOMENTAR

V prilogi C navajamo še nekaj drugih zanimivih ontologij. Diagram C.1 tako predstavlja ontologijo izdelano na celotni množici člankov tujih tiskovnih agencij, ostali pa ločeno za posamezne države (C.2 Hrvaška, C.3 Avstrija, C.4 Nemčija, C.5 Srbija in Črna Gora ter C.6 ZDA).

Pri ontologiji iz vseh tujih člankov opazimo, da je Ontogen našel dobro delitev na prvem nivoju, razdelil je novice po državah oz. tiskovnih agencijah. To med drugim pomeni, da so si države glede poročanja res dovolj različne, da je to dobra delitev. Zanimiva je tudi vsebina naslednjega nivoja. Če na primer pogledamo temo takrat aktualne ptičje gripe, je zanimala le Avstrijce in Nemce, kar je verjetno posledica širjenja bolezni od juga proti severu. Države južno od nas ta tematika ni signifikantno zanimala. Seveda lahko najdemo tudi stalne teme v meddržavnih odnosih, kot npr. avstrijski problemi z našo nuklearno elektrarno in težave na avstrijskem koroškem, problemi s hrvaško mejo in epikontinentalnim pasom, … Tudi v tem diagramu še najdemo nekaj konceptov, ki nas manj zanimajo. Taki so na primer koncepti, katerih ključne besede so ljudje oz. politiki, ampak očitno so bili ti koncepti tako značilni, da jih je Ontogen predlagal kot signifikantne.

Ontologij za ostale države na tem mestu ne bomo podrobno razlagali, saj je prav samo-razlaga bistvena lastnost ontologij. Pri tem pa je pomembna tudi subjektivna ocena bralca, kaj koncepti pravzaprav pomenijo. V glavnem so drugi diagrami razširitev C.1 za dodaten nivo in nekateri koncepti so prav zanimivi, zato spodbujamo bralca, da si jih natančneje ogleda.

S temi rezultati smo zaključili naš prvotni problem nakazan v poglavju "1.5 Motivacija in prispevek dela". Kot rečeno pa je potrebno za primerjavo tem različnih držav dodati še nekaj subjektivnega prispevka, saj se s splošnejšo avtomatsko metodo za primerjavo ontologij nismo ukvarjali.

6 ZAKLJUČEK

V diplomski nalogi smo zasnovali ter v okolju c++ implementirali sistem za gradnjo, uporabo ter evalvacijo lematizatorjev, ki temeljijo na principu *ripple down rules*. V tem delu so natančno opisani vsi zahtevnejši algoritmi, ki samo jih v implementaciji izdelali in uporabili, podana je primerjalna analiza z učnim RDR algoritmom iz [13] na podatkih večjezičnih leksikonov ter prikazana aplikacija na člankih tiskovnih agencij.

Rezultat razvoja programskega sistema, ki je predmet pričujoče diplomske naloge, je lematizator, ki se uporablja kot stopnja predobdelave besedil pri njihovi pripravi za skoraj vse metode odkrivanja znanj. Ostali deli sistema služijo izdelavi lematizatorja in so zasnovani tako, da omogočajo enostavno, hitro, kvalitetno in uporabniku prijazno izdelavo lematizatorjev ter njih ocenjevanje.

Programi in njihova izvorna koda so pod pogoji licence LGPL prosto dostopni. Objavljeni so pod imenom *LemmaGen* na spletnem naslovu *http://kt.ijs.si/software/LemmaGen* ter na diplomskemu delu priloženi zgoščenki. Dodana je tudi vsa potrebna dokumentacija, testni podatki in čim več že izdelanih lematizatorjev za nekaj evropskih jezikov.

Čeprav izdelan sistem zadovoljivo rešuje problem lematizacije, pa ostaja še nekaj izboljšav. Največji izziv za postopke lematizacije posamične besede je, da enaka morfološka oblika lahko izhaja iz več različnih lem. Na ta način je zato nemogoče z gotovostjo izbrati pravo lemo. Za povečanje točnosti predlaganega algoritma bi bilo zato potrebno upoštevati kontekst besede. Eden izmed načinov boljše lematizacije bi bil upoštevanje sosednjih besed iz besedila, kadar so te na voljo. To vprašanje puščamo odprto, dobra rešitev pa bo zelo verjetno še povečala točnost postopka.

PRILOGE

PRILOGA A. MODULI RAZVITEGA SISTEMA

V tej prilogi so razloženi vsi moduli sistema, ki smo ga razvili v okviru te diplomske naloge. Vsa koda je napisana v c++, uporabniški vmesniki pa so izdelani v obliki ukazov in povratnih sporočil v komandni vrstici. Pri implementaciji smo se potrudili, da je knjižnico mogoče prevesti tako v okolju *Windows* kot tudi *linux*, zato lahko rečemo, da je sistem prenosljiv. Izvorna koda je pod licenco LGPL, ki omogoča prosto spreminjanje in uporabo v odprtokodnih in tudi komercialnih aplikacijah. Ker so izdelani moduli zanimivi tudi mednarodno, smo za komunikacijo z uporabnikom uporabili angleški jezik. Naša knjižnica, ki vsebuje vso predstavljeno kodo se imenuje *LemmaGen*.

Ko smo končali razvoj algoritmov in uporabniških vmesnikov zanje, smo ugotovili, da je včasih za uporabnika neprijazno, ker mora skrbeti za sedem ločenih modulov (recimo pri kopiranju ali pošiljanju po elektronski pošti). Tako smo naredili še en uporabniški vmesnik imenovan *LemmaGen*, ki v sebi združuje funkcionalnost vseh ostalih. Izpis pomoči za *LemmaGen* podaja tabela A.1.

TABELA A.1: IZPIS POMOČI MODULA *LEMMAGEN*

```
 1
 2  Wrapper of complete LemmaGen library funcionalty.
 3  Usage: LemmaGen <subproc> [<subproc switches>]
 4
 5    <subproc>              specify subporcess to run, possible choiches:
 6       LemLearn              Learns RDR tree from examples in multext format
 7       LemBuild              Builds lemmatizer data from the rdr tree def. file
 8       Lemmatize             Lemmatizes text and produces new lemmatized file
 9       LemXval               Validates accurancy of learning algorithm
10       LemTest               Calculates lemmatization accurancy from two files
11       LemStat               Calculates statistics about input example data file
12       LemSplit              Splits example file into subsets for validation
13
14    <subproc switches>     try help of the subprocesses for switches definition
15
16    -h, --help             print this help and exit
17        --version          print version information and exit
18
```

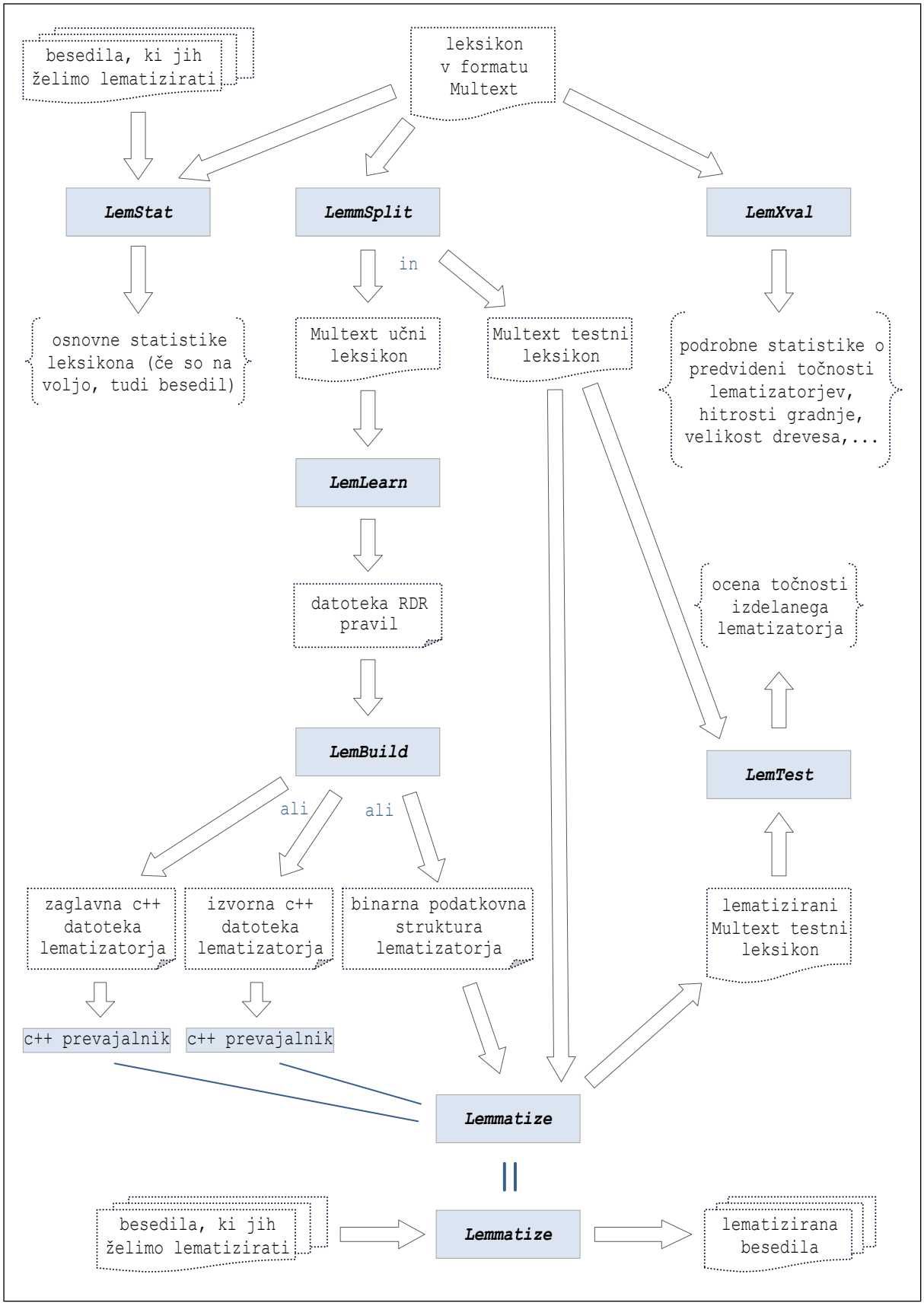


DIAGRAM A.1: SHEMA UPORABE POSAMEZNIH MODULOV

Vloga vmesnika *LemmaGen* je posredovanje ukazov izbranemu podmodulu. Tako je npr. izvajanje ukaza *"LemLearn –v –o outputfile.txt inputfile.txt"* povsem enako kot *"LemmaGen LemLearn –v –o outputfile.txt inputfile.txt*". Uporabnik tako operira le z eno izvršilno datoteko, ki mu za pomoč še kratko opiše vlogo posameznih podmodulov. Izvršilnih datotek za podmodule tako ne potrebuje več, saj vso funkcionalnost prevzame kar *LemmaGen.*

Različni moduli sprejemajo enake oblike datotek zato kar tukaj navajamo njihovo skupno definicijo, na katero se sklicujemo v spodnjih podpoglavjih:

- Besedilna datoteka je poljubna datoteka, ki vsebuje besedilo. Vključuje lahko tako ločila kot tudi presledke, ki jih med branjem ignoriramo. Znaki, ki jih ne prepoznamo kot ločila ter združimo v besede so naslednji: pomišljaj '-', oz. podčrtaj *'_', {0-9}, {a-z}, {A-Z},* ter znaki z *ASCII* kodo večjo od 127.
- Datoteko Multext smo že razložili v poglavju 5.1.1, tukaj je kratek povzetek. Vsaka vrstica ima enak format, sestavljena pa je iz treh "besed" ločenih z dvema tabulatorjema. Znaki od začetka vrstice do prvega tabulatorja predstavljajo morfološko obliko besede. Preskočimo tabulator. Znaki od tu do naslednjega tabulatorja predstavljajo lemo besede. Prostor za tem tabulatorjem ter do konca vrstice pripada morfološkemu opisu besede. Ker zadnjega stolpca ne uporabljamo je lahko ta tudi prazen oz. vsebuje poljuben niz. Skrajšano vrstica izgleda: $v = (morf.oblika) \cdot [tab] \cdot (lema) \cdot [tab] \cdot (morf.opis)$.
- Datoteka RDR je datoteka, ki vsebuje RDR pravila. Ta format datoteke smo orisali že v poglavju 1.4 "RDR v domeni lematizacije", opisali v 3.2 "Določitev vhodne datoteke" ter zelo natančno definirali v 3.3 "Struktura algoritma ter implementacija" Primere notacij te datoteke lahko najdete v primeru 3.1.
- Zaglavno, izvorno ter binarno RDR datoteko generira modul *LemBuild* in je ni mogoče izdelati ročno.

Tu povejmo le še da je uporaba opcij oz stikal pri klicu vmesnikov neodvisna od njihovega vrstnega reda. V primeru že obstoječe izhodne datoteke pa moduli vedno vprašajo, če jo lahko prepišejo.

Diagram A.1 podaja osnovno predstavo o tem, kako moduli med sabo sodelujejo in kakšne vhode zahtevajo oz. izhode generirajo. Na shemi je prikazan tok podatkov za vse korake od učenja RDR drevesa do testiranja točnosti. Sledeča poglavja natančno opisujejo uporabniške vmesnike vseh sedmih izdelanih modulov.

RDR Lematizator

Modul, ki se ukvarja z lematizacijo (poglavje 2) smo poimenovali *Lemmatize*. Kot vhod sprejema tako besedilne datoteke kot tudi datoteke Multext.

Z opcijo *<format>* izbiramo, kako bomo lematizirali. V primeru izbire *"text"* algoritem obravnava vhodno datoteko kot besedilno in temu primerno na izhod zapiše lematizirano besedilo. Če pa izberemo *"wpl",* bo algoritem vedno lematiziral le prvo besedo v vrstici, na izhod pa bo zapisal nespremenjeno prvo besedo, tabulator, lematizirano prvo besedo, tabulator,preostanek vrstice iz vhodne datoteke. V primeru uporabe opcije *<delmt>* modulu podamo nek znak. Algoritem nato zamenja vsa ločila v izhodni datoteki (*"text"*) ter tabulatorje (*"wpl"*) z izbranim znakom. Natančno definicijo kako se podajajo opcije prikazuje tabela A.2, primer A.1 pa oriše tipično uporabo tega modula.

Z opcijo *<datafile>* podamo modulu binarno datoteko RDR pravil, ki jih modul uporablja za lematizacijo. V kolikor tega ne storimo, se uporabljajo privzeta pravila. Če nismo izdelali lematizatorja za določen jezik in ga prevedli, so privzeta pravila prazna in lematizacija ne naredi nobene spremembe.

TABELA A.2: IZPIS POMOČI MODULA *LEMMATIZE*

```
 1
 2  Lemmatizes text and produces new lemmatized file
 3  Usage: Lemmatize [-f <format>] [-l <datafile>] [-d <delmt>] <infile> <outfile>
 4
 5    -f, --format <format>  specify input/output file format (default = text)
 6       text                  outfile = infile just words are lematized
 7       wpl                   word per line, outfile line = word + tab + lemma
 8    -l, --lang <datafile>  binary language data file that defines lemmatization
 9                           rules, if not specified it lematizes by default rules
10                           that were build by compilation. If you did't make
11                           your own compilation then rules are empty.
12    -d, --delimit <delmt>  removes punctuations from <outfile>, one or more
13                           punct. together are replaced with single <delmt>
14
15    <infile>               input file (file to be lemmatized)
16    <outfile>              output file (lemmatized file)
17
18    -v, --verbose          verbose messages
19    -h, --help             print this help and exit
20        --version          print version information and exit
21
```

PRIMER A.1: ZGLED TIPIČNEGA KLICA MODULA *LEMMATIZE*

```
 1  Lemmatize -d " " -l slo.bin notlem.txt lemmatized.txt -v
```

```
 1  chosen options:
 2           input file: notlem.txt
 3          output file: lemmatized.txt
 4   language data file: slo.bin
 5                format: plain text
 6               verbose: yes
 7     output delimiter: ' '
 8  processing...
 9    Lemmatized 21388 lines, 1455294 words, 729.062ns per word.
10  finished successfully
```

GRADNJA LEMATIZATORJEV

Naloga modula *LemBuild* (poglavje 3) je gradnja strukture lematizatorja iz datoteke RDR pravil. Možnih izhodnih datotek je več in so napisane spodaj. Izpis pomoči modula podaja tabela A.3, tipičen primer uporabe pa primer A.2.

Opcije modula se nanašajo v glavnem na izbiro izhoda *<type>*. Odločimo se lahko za binarno izhodno datoteko *"bin"*, c++ zaglavno datoteko *"head"*, datoteko izvorne kode c++ *"src"* ali pa strukturo izpisano v človeku prijazni obliki *"human"*. Slednja možnost se uporablja zgolj za pregled nad notranjo strukturo podatkov lematizatorja, predzadnji dve pa za izdelavo lematizatorja, katerega privzeta pravila so ta, ki jih tukaj prevajamo. Najpogosteje se uporablja možnost *"bin"*, ki vrne strukturo primerno za direktni uvoz v *Lemmatize*.

Opcija *<language>* v izhodno strukturo zapiše podatek o jeziku, uporabi pa se tudi kot privzeto ime za izhodno datoteko. Kadar generiramo lematizator v obliki izvorne kode c++, je pomembna tudi opcija *<skeldir>,* s katero modulu podamo pot do izvornih datotek tega sistema (LemmaGen knjižnica). Le tako lahko sestavi delujočo izvorno kodo lematizatorja z dodanimi privzetimi pravili lematizacije.

TABELA A.3: IZPIS POMOČI MODULA *LEMBUILD*

```
 1
 2  Builds lemmatizer structure from the rdr tree definition file
 3  Usage: LemBuild [-t <type> [-s ...]] [-l <language>] [-o <outfile>] <infile>
 4
 5    -t, --type <type>      specify output type (default = bin), valid options:
 6       bin                   binary file that can be imported into Lemmatize
 7       head                  header file for libraries that are using LemmaGen
 8       src                   class file that can be compiled to Lemmatize,
 9                             must specify directory where skeleton files reside
10       human                 human readable lemmatizer structure
11    -s, --skelet <skeldir> directory where program can find skeleton files
12    -l, --lang <language>  specify language for output file naming and
13                             lemmatizer definition (default is 'unknown-lang')
14
15    -o, --out <outfile>    output file (default is <language>.<type>)
16    <infile>               input file (rdr tree definition file)
17
18        --stat             print statistics about pareed tree
19    -v, --verbose          verbose messages
20    -h, --help             print this help and exit
21        --version          print version information and exit
22
```

Pri uporabi pride pogosto prav tudi opcija *"--stat"*, ki izpiše statistiko drevesa pred optimizacijo in po njej. Tako dobimo pregled nad tem, kako se je drevo spremenilo. V primeru A.2 so zaradi kompaktnejše predstavitve določene vrstice izhoda izpuščene (označene z "…").

PRIMER A.2: ZGLED TIPIČNEGA KLICA MODULA *LEMBUILD*

```
LemBuild rdr.tree -v --lang slo --stat
```

```
chosen options:
         input file: rdr.tree
        output file: slo.bin
 language specifier: slo
        output type: binary
            verbose: yes
         statistics: no
processing...
  Starting to parse input file...
  Parsing successfull in 0.27 s.
  Tree statistics: ------------------------------------------------------------
    Numer of nodes: 35851
      Internal:     11874
      Leaf:         23977
    Average depth:  5.92
    Avg. branching: 3.02
    Distribution of internal and leaf nodes by depth:
      |Depth|Internal|  Leaf |Together|
      |   0 |      1 |     0 |      1 |
...  ...
      |  14 |      0 |     8 |      8 |
    Distribution of first 10 most frequent branchings:
      Branc      0     1     2     3     4     5     6     7     8     9    10
      Numbr  23977  2685  4724  1630   946   573   382   242   151   113   107
  ------------------------------------------------------------------------------
  Starting optimization...
  Optimization complete in 0.07 s.
  Tree statistics: ------------------------------------------------------------
...  ...
  ------------------------------------------------------------------------------
  Starting lemmatizer generation...
  Lemmatizer created in 0.14 s.
  Generating output file...
  File generated successfully in 0.01 s.
  Time needed altogether 0.49 s.
finished successfully
```

UČENJE RDR PRAVIL

Modul učenja pravil iz učnih primerov (poglavje 4) se imenuje *LemLearn*. Vhod je datoteka Multext, izhod pa datoteka RDR.

V tabeli A.4 lahko vidimo kako se *LemLearn* uporablja. Najpomembnejša opcija je *<alg>* s katero izbiramo želeni algoritem učenja. Na voljo imamo prekrivni algoritem *"cover"*, ki je privzet in originalni RDR algoritem "*rdr"*. Poleg tega opcija *<format>* omogoča še izbiro enega izmed šestih oblik izpisa pravil v izhodno datoteko.

V primeru A.3 vidimo tudi uporabo opcije *"—stat"*, ki podobno kot pri modulu *LemBuild*, omogoča podrobnejši prikaz lastnosti drevesa. S to statistiko dobimo že prvo oceno kvalitete drevesa, saj je podana tudi lematizacijska točnost na učnih podatkih.

TABELA A.4: IZPIS POMOČI MODULA *LEMLEARN*

```
 1
 2  Learns RDR tree from file of examples formated in multext standard
 3  Usage: LemLearn [-f <format>] [-a <alg>-] -o <outfile> <infile> [<infile2>,..]
 4
 5    -f, --format <format>  specify output tree formating (default=4)
 6       1,2,3,4,5,6           6 possible formats, try them to find the one that
 7                             suits your needs best, all are valid for LemBuild
 8    -a, --algorithm <alg>  algorithm to be used in learning (default=cover)
 9       cover                 newly developed, faster and usually more accurate
10       rdr                   original that strictly folows rdr metodology
11
12    -o, --out <outfile>    output file (human readable rdr tree)
13    <infile>               input file (multext, one example per line)
14                             format of each line: 'word[tab]lemma[tab]*'
15    <infile2>              more input files = equal to the one containg all
16                           the data
17
18        --stat             print statistics about pareed tree
19    -v, --verbose          verbose messages
20    -h, --help             print this help and exit
21        --version          print version information and exit
22
```

PRIMER A.3: ZGLED TIPIČNEGA KLICA MODULA *LEMLEARN*

```
1  LemLearn multext_slo.tbl -o rdr.tree -v --stat
```

```
 1  chosen options:
 2        input file(s): multext_slo.tbl
 3          output file: rdr.tree
 4               format: 4
 5            algorithm: new covering
 6              verbose: yes
 7           statistics: yes
 8  processing...
 9    Proccessing file '..\..\..\data\multext\ascii_dif_wfl-sl.tbl'.
10      Imported 557970 lines, 14025810 bytes = 13.38 MB. Current word count 557970
11      Time needed for read 1.03 s.
12      Import together 2.82 s.
13    Words sorted in 2.49 s.
14    Learning completed in 1.20 s.
15    Tree statistics:
16      Nodes       35851
17        Leaves    23977
18        Internal  11874
19        Max Depth 14
20        Branching 3.02
21      Words       557970
22        Positives 542880
23        Negatives 15090
24        Accurancy 97.2956%
25        Error     2.7044%
26        Avg Depth 4.4891
27    File generated successfully in 0.50 s.
28  finished successfully
```

PREČNO PREVERJANJE TOČNOSTI

Modul za preverjanje točnosti naučenih lematizatorjev, *LemXval* je med vsemi najkompleksnejši. Njegov vhod je datoteka Multext, izhod pa tekstovna datoteka z zapisanimi statistikami, ki jih lahko nato obdelujemo s poljubnim tekstovnim ali tabelaričnim programom.

V tem modulu imamo poleg običajnega obveznega podatka (vhodne datoteke), obvezen še podatek *<split>.* Ta definira, na koliko podmnožic se razbije učna množica pri prečnem preverjanju (tip "*shallow*" ali "*deep*") oz. koliko procentov ima testna množica v enkratnem preverjanju (tip "*perc*" ali "*percdeep*").

Opcija *<type>* določi način bomo razbili množico primerov na učno in testno množico. "*shallow*" in "*deep*" definirata prečno preverjanje. "*deep*" razbitje tako, da dobimo pesimistično oceno točnosti (opisano v poglavju 5.1.2), pri "*shallow*" pa približno realno oceno. Analogno velja tudi za "*perc*" in "*percdeep*", le da se tukaj množica razdeli le enkrat tako, da je *<split>* procent primerov v testni množici.

TABELA A.5: IZPIS POMOČI MODULA *LEMXVAL*

```
 1
 2  Validates quality of learning according to chosen options:
 3     if type of split is shallow or deep then it uses cross validation else
 4     it just tests test set on the rules that were learned on the train set
 5
 6  Usage: LemXval -k <split> [-t <type>] [-a <alg>] [-o <outfile>] <infile>
 7
 8    -k, --split <split>    type 's','d': number of required subsets (cross)
 9                           type 'p','r': percent of words to be in test set
10    -t, --type <type>      specify type of split (default = deep), valid:
11       s|shallow             split set according to example lines
12       d|deep                split set according to example contents, all examp.
13                             with same word and lemma are in the same subset
14       p|perc                split on 2 subsets using percent for test set
15       r|percdeep            combination of percent and deep functionality
16
17    -a, --algorithm <alg>  algorithm to be used in learning (default=cover)
18       cover                 newly developed, faster and usually more accurate
19       rdr                   original that strictly folows rdr metodology
20       both                  both stated above
21
22    -e, --extend           calculate even more statistics
23
24    -o, --out <outfile>    output file (print statistics to the file
25    <infile>               input file (multext format)
26
27    -v, --verbose          verbose messages
28    -h, --help             print this help and exit
29        --version          print version information and exit
30    -s, --silent           print just basic information
31
```

PRIMER A.4: ZGLED TIPIČNEGA KLICA MODULA *LEMXVAL*

```
LemXval multext_slo.tbl -k 5 -v -a both
```

```
chosen options:
         input file: multext_slo.tbl
        output file: none
      type of split: shallow (5 sets)
          algorithm: new covering & original rdr
           extended: no
             silent: no
            verbose: yes
processing...
  Test started...  multext_slo.tbl
    Time needed to open 0.00 s.
    Imported 557970 lines, 14025810 bytes = 13.38 MB. Current word count 557970
    Time needed for read 1.05 s.
    Insertion to array of words 1.10 s.
    Import together 2.87 s.
  Spliting set to K-fold. K = 5, deep split = false.
  Randomized sets. Time 0.16 s.
  Original set of 557970 words splited in 0.10 s into next subsets:
    111629,111209,111964,111212,111956
  Filled new tables. Time 0.06 s.
  Sorting 557970 words.
  Words sorted in 2.54 s.
  Test started...
    Set 1 - cover:[....] 94.30% 97.42% 4.03s rdr:[....] 92.48% 95.17% 38.23s
    Set 2 - cover:[....] 94.26% 97.39% 5.72s rdr:[....] 92.52% 95.17% 54.62s
    Set 3 - cover:[....] 94.45% 97.40% 5.25s rdr:[....] 92.72% 95.12% 66.30s
    Set 4 - cover:[....] 94.41% 97.39% 4.95s rdr:[....] 92.56% 95.14% 74.78s
    Set 5 - cover:[....] 94.35% 97.42% 5.27s rdr:[....] 92.58% 95.15% 68.97s
    Average for cover  (on test set): 94.3546%
    Average for rdr    (on test set): 92.5720%
    Average for cover (on train set): 97.4052%
    Average for rdr   (on train set): 95.1507%
  Time needed for all tests 329.51 s.
  Time needed altogether 336.34 s.
finished successfully
```

Z opcijo *<alg>* izberemo učni algoritem, s katerim želimo generirati drevesa. Če prečno preverjanje poganjamo večkrat, pride prav še opcija *"-s"*, ki omogoči uporabo skript brez nadležnega izpisovanja poteka algoritma. V tabeli A.5 lahko vidite izpisano pomoč modula *LemXval*, v primeru A.4 pa tudi zgled njegove uporabe.

STATISTIKA LEKSIKONOV

Modul *LemStat* je enostaven, njegova naloga pa je izračun osnovnih statistik nekega leksikona (vhod je torej datoteka Multext) in po želji uporabnika zapis v izhodno datoteko. V tabeli A.6 lahko vidimo izpis pomoči tega modula..

Če imamo na voljo tudi datoteko z besedilom za katerega nas zanima recimo pokritost besed z besedami iz leksikona, potem uporabimo opcijo *<textfile>*. Statistike, ki jih modul vrača so nazorno prikazane v primeru A.5 zato jih tu ne bomo podrobneje razlagali.

TABELA A.6: IZPIS POMOČI MODULA *LEMSTAT*

```
 1
 2  Calculates some basic statistics about input example data file
 3  Usage: LemStat [-o <outfile>] [-t <textfile>] <infile> [<infile2>,...]
 4
 5    -t, --text <textfile>  some additional statistics that can be derived
 6                           from combination with text file (words matching %)
 7
 8    -o, --out <outfile>    output file, if not specified statistics are
 9                           writen to screen
10    <infile>               input file (multext, one example per line)
11                             format of each line: 'word[tab]lemma[tab]*'
12    <infile2>              more input files, each is treated separatelly
13
14    -v, --verbose          verbose messages
15    -h, --help             print this help and exit
16        --version          print version information and exit
17
```

PRIMER A.5: ZGLED TIPIČNEGA KLICA MODULA *LEMSTAT*

```
1  LemStat multext_slo.tbl -t texts.txt -v
```

```
 1  chosen options:
 2        input file(s): multext_slo.tbl
 3          output file: none -> screen
 4            text file: texts.txt
 5              verbose: yes
 6  processing...
 7    Proccessing file 'multext_slo.tbl '.
 8      Time needed to open 0.00 s.
 9      Imported 557970 lines, 14025810 bytes = 13.38 MB. Current word count 557970
10      Time needed for read 1.06 s.
11      Insertion to array of words 1.15 s.
12      Import together 2.93 s.
13      Calculating statistics for file 1...
14        -------------------------: ------
15                      file entrys: 557970
16                            words: 198507
17                           lemmas: 16389
18                            forms: 2083
19        -------------------------: ------
20        avg. difr. words per lemma: 12.6331
21        avg. difr. lemmas per word: 1.0430
22        -------------------------: ------
23                words in text file: 1455294
24         diffr. words in text file: 89415
25        difference text percentage: 6.14%
26        -------------------------: ------
27             words in lexicon file: 557970
28        diffr. words in lexi. file: 198116
29         diffr. lexicon percentage: 35.51%
30        -------------------------: ------
31          same in lexicon and text: 34114
32         percent of lexicon coverd: 20.67%
33            percent of text coverd: 83.72%
34        -------------------------: ------
34    Time needed altogether 15.16 s.
36  finished successfully
```

PRIPRAVA UČNIH IN TESTNIH MNOŽIC

Modul *LemSplit* rešuje problem izdelave učnih in testnih množic iz leksikona. Vhod je datoteka Multext, izhod dve ali več datotek Multext. Primer A.6 podaja tipično uporabo tega modula, tabela A.7 pa izpis pomoči.

Opciji *<split>* in *<type>* se obnašata popolnoma enako kot v modulu *LemXval*. Nestandardni obvezni podatek tega modula je še *<outfilemask>* s katerim specificiramo, kakšna imena bodo imele nove datoteke. Imena dobi algoritem tako, da zamenja '# ' v *<outfilemask>* s oznako trenutne generirane datoteke. Če '#' ne najde, potem doda oznako kar na konec imena datoteke.

TABELA A.7: IZPIS POMOČI MODULA *LEMSPLIT*

```
 1
 2  Splits example data file into smaller subsets for cross validation
 3  Usage: LemSplit -k <split> [-t <type>] <infile> <outfilemask>
 4
 5    -k, --split <split>    type 's','d': number of required subsets (cross)
 6                           type 'p','r': percent of words to be in test set
 7    -t, --type <type>      specify type of split (default = deep), valid:
 8       s|shallow             split set according to example lines
 9       d|deep                split set according to example contents, all examp.
10                             with same word and lemma are in the same subset
11       p|perc                split on 2 subsets using percent for test set
12       r|percdeep            combination of percent and deep functionality
13
14    <infile>               input file (multext format)
15    <outfilemask>          output file mask ('#' replaced with sequence number)
16
17    -v, --verbose          verbose messages
18    -h, --help             print this help and exit
19        --version          print version information and exit
20
```

PRIMER A.6: ZGLED TIPIČNEGA KLICA MODULA *LEMSPLIT*

```
 1  LemSplit  -t percdeep -k 25 multest_slo.tbl out_#.tbl -v
```

```
 1  chosen options:
 2           input file: multest_slo.tbl
 3     output file mask: out_#.tbl
 4        type of split: percent deep (25% in test set)
 5              verbose: yes
 6  processing...
 7    Proccessing file 'multest_slo.tbl'.
 8      Time needed to open 0.00 s.
 9      Imported 557970 lines, 14025810 bytes = 13.38 MB. Current word count 557970
10      Time needed for read 1.03 s.
11      Insertion to array of words 1.10 s.
12      Import together 2.84 s.
13    Generating file 'out_train.tbl' ...
14    Generating file 'out_test.tbl' ...
15    Time needed altogether 5.69 s.
16  finished successfully
```

TESTIRANJE TOČNOSTI LEMATIZACIJE

Zadnji modul se uporablja za testiranje točnosti lematiziranega leksikona in se imenuje *LemTest*. Vhod modula sta dve datoteki oblike Multext: originalni leksikon in originalni leksikon, ki je bil lematiziran z modulom *Lemmatize* in opcijo *"wpl"*.

Pri opisu *Lemmatize* smo omenili, da algoritem zapiše lematizirano besedo na drugo mesto, takoj za nelematizirano. Oblika tako ostane zelo podobna leksikonu, le da zdaj v drugem stolpcu niso resnične leme, ampak tiste, ki smo jih dobili z lematizacijo. Naloga za *LemTest* je tako preprosta, saj le primerja isto ležne besede iz drugih stolpcev danih datotek. Tabela A.8 prikazuje izpis pomoči, primer A.7 pa zgled uporabe modula *LemTest*

TABELA A.8: IZPIS POMOČI MODULA *LEMTEST*

```
 1
 2  Calculates and outputs lemmatization accurancy from two files
 3  Usage: LemTest <correctfile> <lemmatizedfile>
 4
 5    <correctfile>           usually file containing test set that
 6                             learning algorithm didn't consult
 7    <lemmatizedfile>       <correctfile> that was lemmatized
 8
 9    -v, --verbose          verbose messages
10    -h, --help             print this help and exit
11        --version          print version information and exit
12
13    NOTE:
14      Both files are in approximate multext format. Each word in separate line
15      in format 'word[tab]lemma[tab]*'. <correctfile> can be created using
16      LemXval to split larger example set. <lemmatizedfile> can be derived
17      from <correctfile> lemmatizing it with Lemmatize option '-t wpl'.
18
```

PRIMER A.7: ZGLED TIPIČNEGA KLICA MODULA *LEMTEST*

```
 1  LemTest multext_slo.tbl multext_slo_lemmatized.tbl
```

```
 1  chosen options:
 2         correct file: multext_slo.tbl
 3      Lemmatized file: multext_slo_lemmatized.tbl
 4              verbose: no
 5  processing...
 6  findings:
 7      correct = 537687
 8        wrong = 20283
 9    accurancy = 96.3649%
10        error = 3.6351%
11  finished successfully
```

PRILOGA B. VSEBINA ELEKTRONSKIH PRILOG

Vse spodaj opisane elektronske priloge je moč najti na zgoščenki, ki je priložena temu delu. Dostopne pa so tudi na spletnem naslovu *http://kt.ijs.si/software/LemmaGen,* kjer boste našli tudi vse informacije v zvezi z morebitnim nadaljnjim razvojem sistema. Tabela B.1 prikazuje datotečno organizacijo elektronskih prilog, B.2 pa verzije orodji, ki smo jih uporabljali in so s prilogami skladna.

TABELA B.1: STRUKTURA UREDITVE ELEKTRONSKIH PRILOG

lokacija	opis
./	
Data/	primeri za testiraje delovanja modulov sistema
|--> lemmatizer/*	izdelani lematizatorji
|--> multex/*	leksikoni
|--> news/*	agencijske novice
|--> ontology/*	ontologije
`--> rdrtree/*	rdr drevesa
Documents/	besedilo tega diplomskega dela v pdf formatu
LemmaGen/	knjižnjica in moduli sistema LemmaGen
|--> binary/	izvršilne datoteke…
| |--> linux/*	…za okolje linux
| | `--> object/*	objekti za projekt Makefile
| `--> win32/	…za okolje Windows
| |--> debug/*	testna verzija izvršilnih datotek
| | `--> object/*	objekti za projekt VisualStudio/debug
| `--> release/*	končna verzija izvršilnih datotek
| `--> object/*	objekti za projekt VisualStudio/release
|--> project/	projekta za prevajanje kode
| |--> makefile/Makefile	projekt za prevajanje kode v okolju linux
| `--> visualstudio/*	projekti za razvoj sistema v okolju Windows
|--> source/	izvorna koda razvitega sistema
| |--> definition/*	defin. datoteke leksikalne in sintaksne analize
| |--> header/*	zaglavne datoteke jedra sistema
| |--> interface/*	zaglavna ter izvorne datoteke za vse vmesnike
| |--> main/*	za vsak modul svoja datoteka s kratko main funk.
| `--> source/*	izvorne datoteke jedra sistema
`--> support/	dodatki uporabljeni pri razvoju sistema
`--> lex-bison/*	izvršilne in izvorne dat. orodji Flex in Bison
OntoGen/*	izvršilne datoteke orogja OntoGen

TABELA B.2: PROGRAMSKA ORODJA UPORABLJENA PRI IZDELAVI DIPLOMSKE NALOGE

naziv	verzija	priloženo diplomi
Microsoft Visual C++ 2005	8.0.50727.762	ne
GNU Make	3.81	ne
OntoGen	2.0.0	da
Flex++	2.3.8-7	da
Bison++	1.21.8	da

PRILOGA C. ONTOLOGIJE

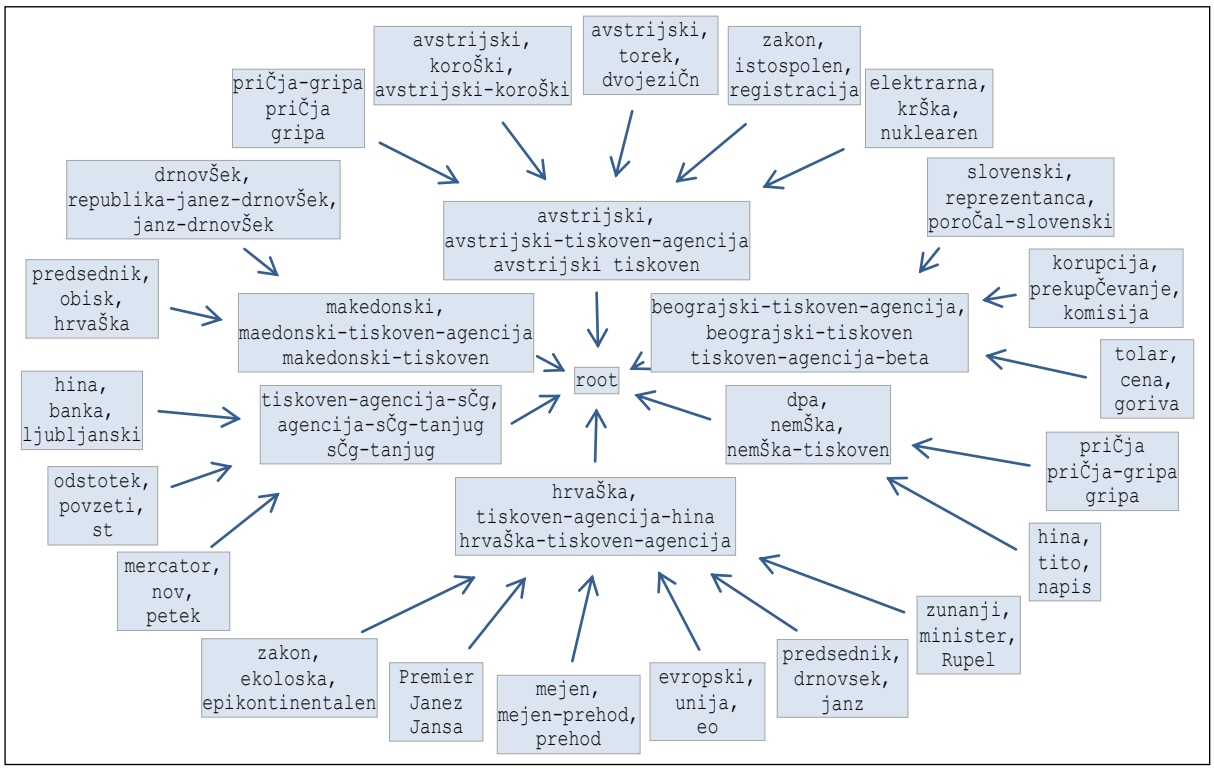


DIAGRAM C.1: ONTOLOGIJA IZ BESEDIL VSEH TUJIH AGENCIJSKIH ČLANKOV

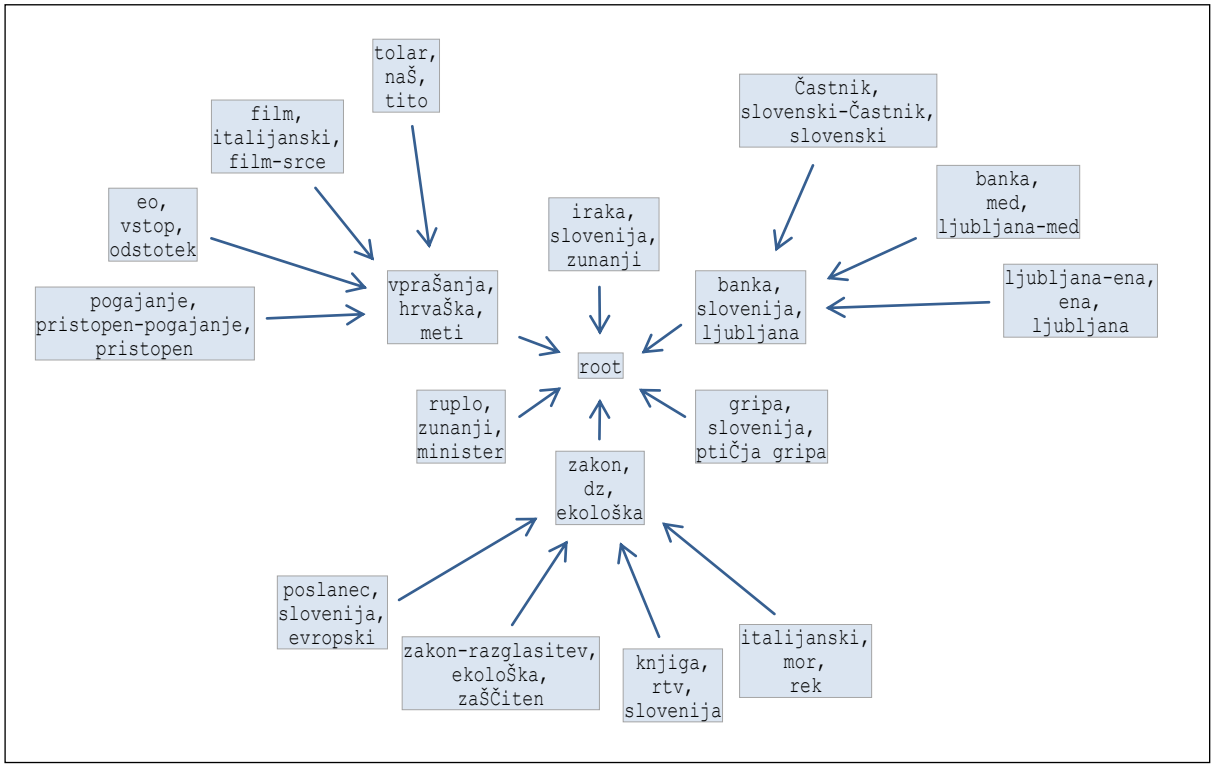


DIAGRAM C.2: ONTOLOGIJA IZ BESEDIL HRVAŠKIH AGENCIJSKIH ČLANKOV

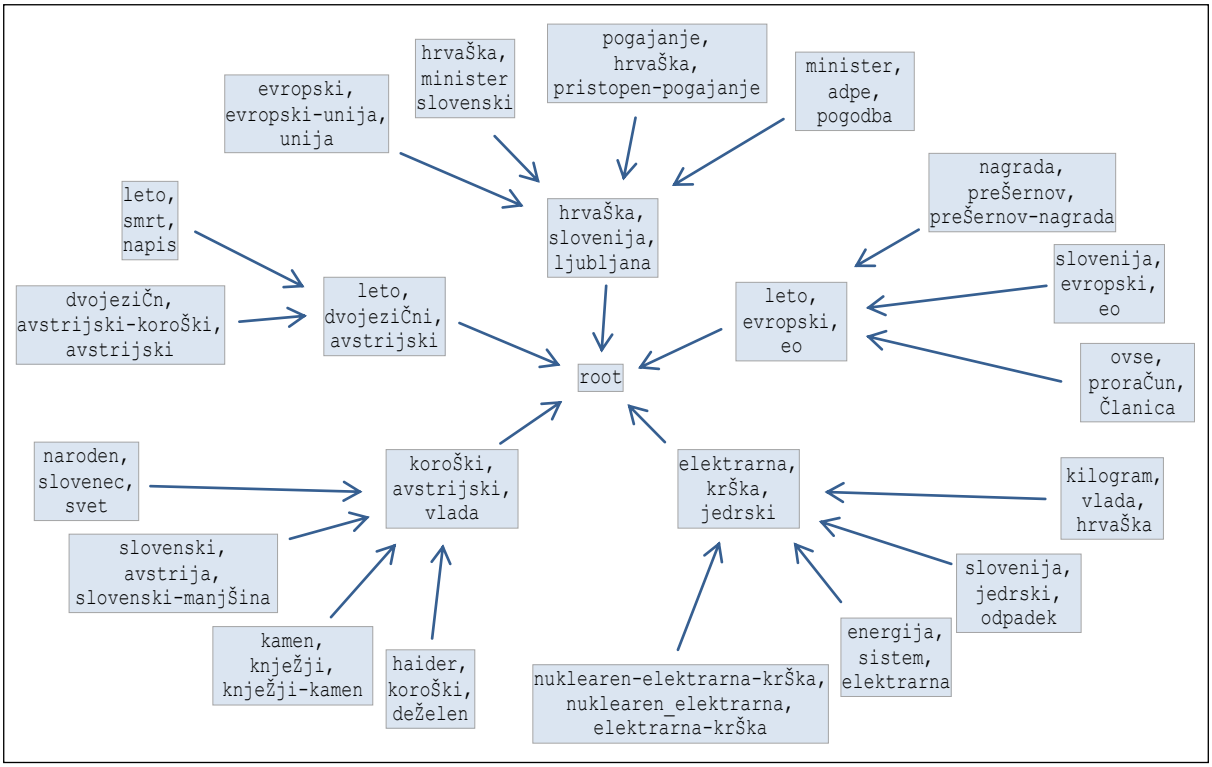


DIAGRAM C.3: ONTOLOGIJA IZ BESEDIL AVSTRIJSKIH AGENCIJSKIH ČLANKOV

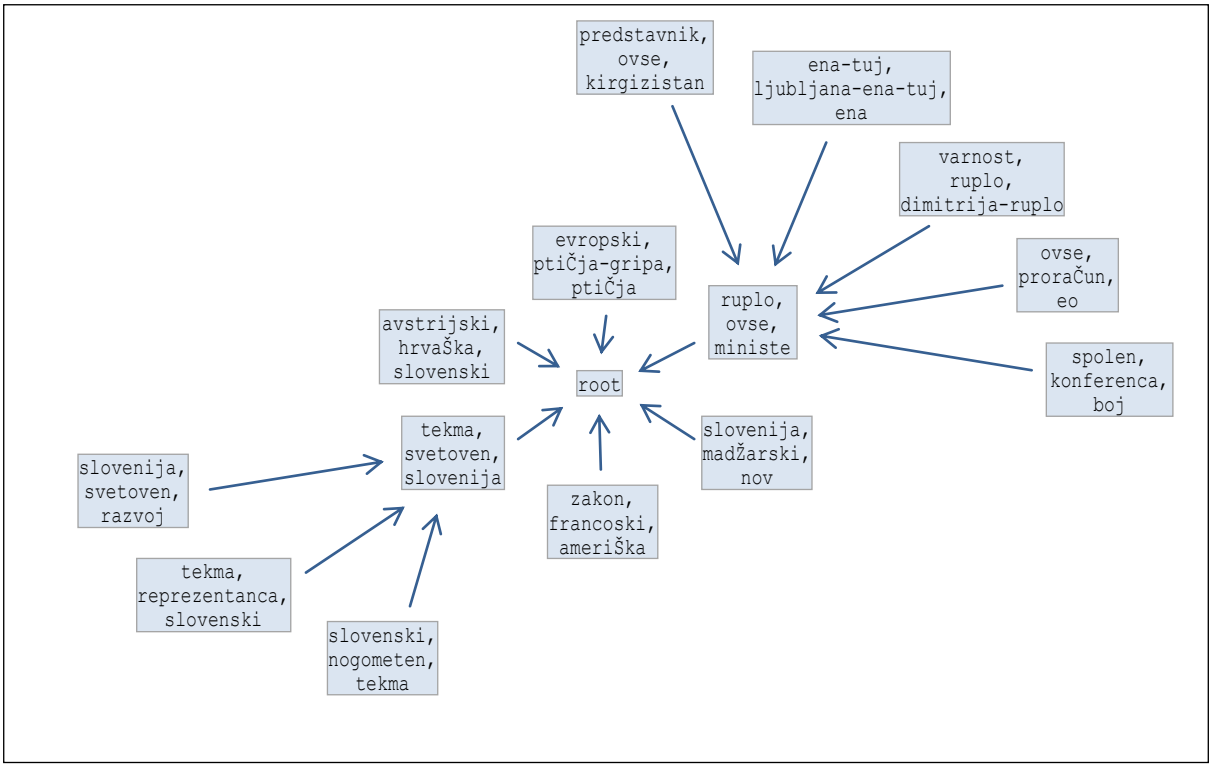


DIAGRAM C.4: ONTOLOGIJA IZ BESEDIL NEMŠKIH AGENCIJSKIH ČLANKOV

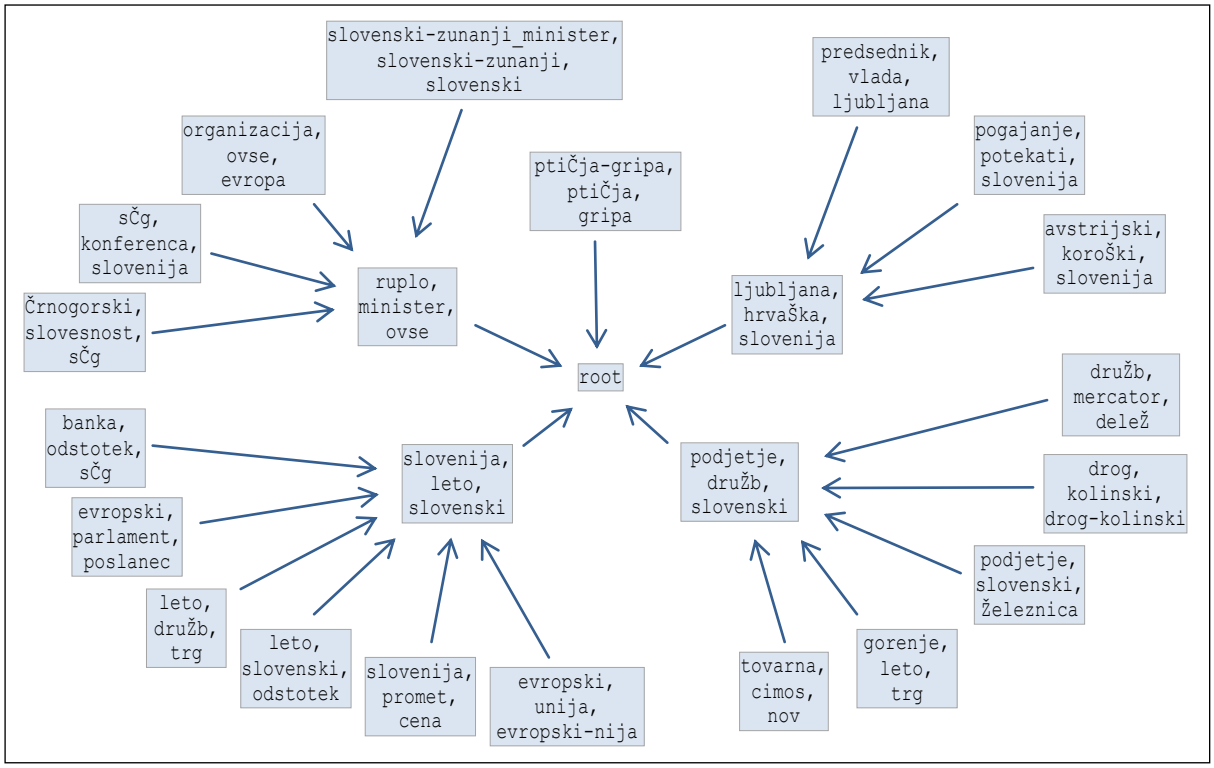


DIAGRAM C.5: ONTOLOGIJA IZ BESEDIL AGENCIJSKIH ČLANKOV SRBIJE IN ČRNE GORE

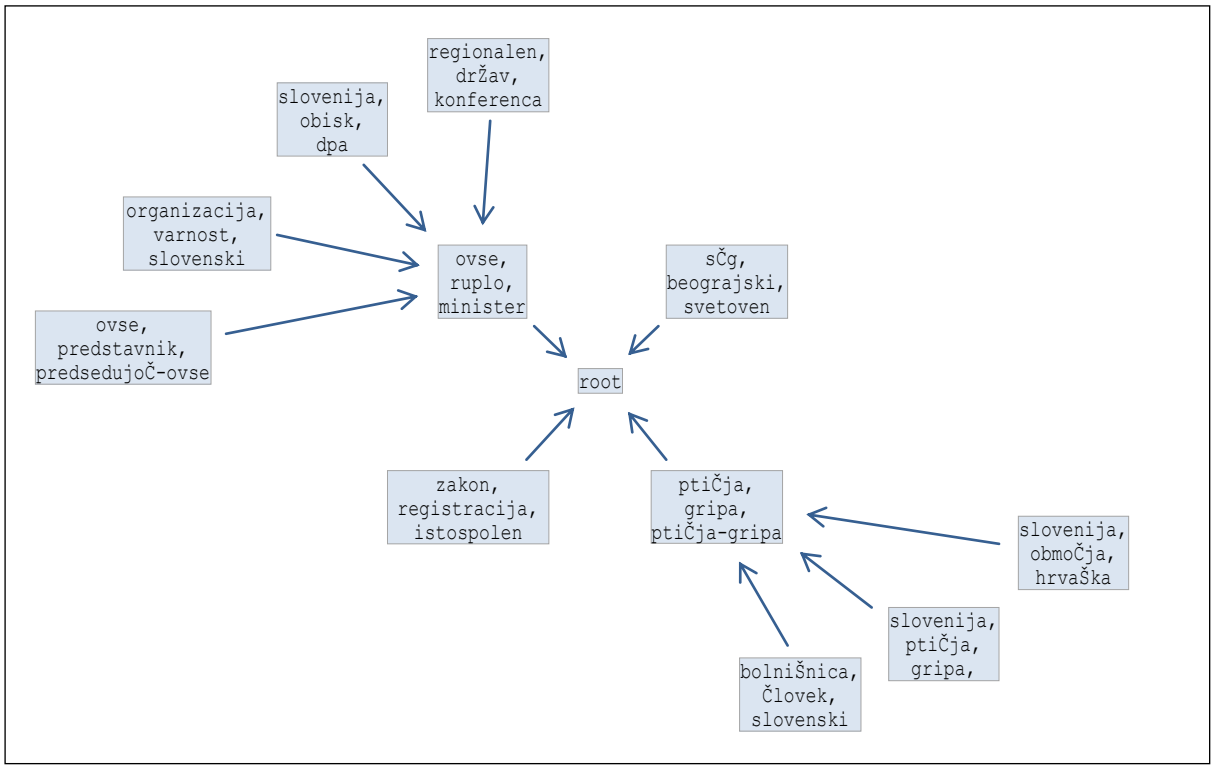


DIAGRAM C.6: ONTOLOGIJA IZ BESEDIL AGENCIJSKIH ČLANKOV ZDA.

PRILOGA D. SEZNAMI

SEZNAM PRIMEROV

SEZNAM DIAGRAMOV

Seznam kode

Seznam tabel

LITERATURA

[1] **Beth, L.J.** Development of a stemming algorithm. *v Mechanical Translation and Computational Linguistics.* zv. 11, str. 22–31, 1968.

[2] **Compton, P. in Jansen, R.** A philosophical basis for knowledge acquisition. *v Proceedings of the 3rd European Knowledge Acquisition for Knowledge Based Systems Workshop.* str. 75-89, 1989.

[3] **Compton, P. in Jansen, R.** Knowledge in Context: a strategy for expert system maintenance. *v Proceedings of the 2nd Australian Joint Artificial Intelligence Conference.* str. 292–306, 1988.

[4] **Erjavec, T. Džeroski, S.** Machine Learning of Morphosyntactic Structure: Lemmatising Unknown Slovene Words. *v Applied Artificial Intelligence.* zv. 18(1), str. 17-40, 2004.

[5] **Erjavec, T.** MULTEXT-East Version 3: Multilingual Morphosyntactic Specifications, Lexicons and Corpora. *v Proceedings of the 4th International Conference on Language Resources and Evaluation LREC-2004.* str. 1535-1538, 2004.

[6] **Fortuna, B., Mladenić, D. in Grobelnik, M.** Semi-automatic Data-driven Ontology Construction System. *v Proceedings of the 9th International multi-conference Information Society IS-2006.* zv. A, str. 217-220, 2006.

[7] **Ide, N. in Véronis, J.** MULTEXT: Multilingual Text Tools and Corpora. *v Proceedings of the 15th conference on Computational Linguistics.* zv. 1, str. 588-592, 1994.

[8] **Mitchell, T.** *Machine Learning.* s.l. : McGraw Hill, 1997.

[9] **Mladenić, D.** Automatic Word Lemmatization. *v Proceedings of the 5th International Multi-Conference Information Society IS-2002.* zv. B, str.153-159, 2002.

[10] **Mladenić, D.** Combinatorial Optimization in Inductive Concept Learning. *v Proceedings of 10th International Conference on Machine Learning ICML-1993.* str 205-211, 1993.

[11] **Mladenić, D.** Learning Word Noramlization Using Word Suffix and Context from Unlabeled Data. *v Proceedings of 19th International Conference on Machine Learning ICML-2002.* str. 427-434, 2002.

[12] **Plisson, J., Lavrač, N. in Mladenić, D.** A rule based approach to word lemmatization. *v Proceedings of the 7th International Multi-Conference Information Society IS-2004.* zv. C, str. 83-86, 2004.

[13] **Plisson, J., in drugi.** Ripple Down Rule Learning for Automated Word Lemmatisation. Poslano v objavo, 2007.

[14] **Porter, M.F.** An Algorithm for Suffix Stripping. *v Program.* zv. 14(3), str. 130−137, 1980.